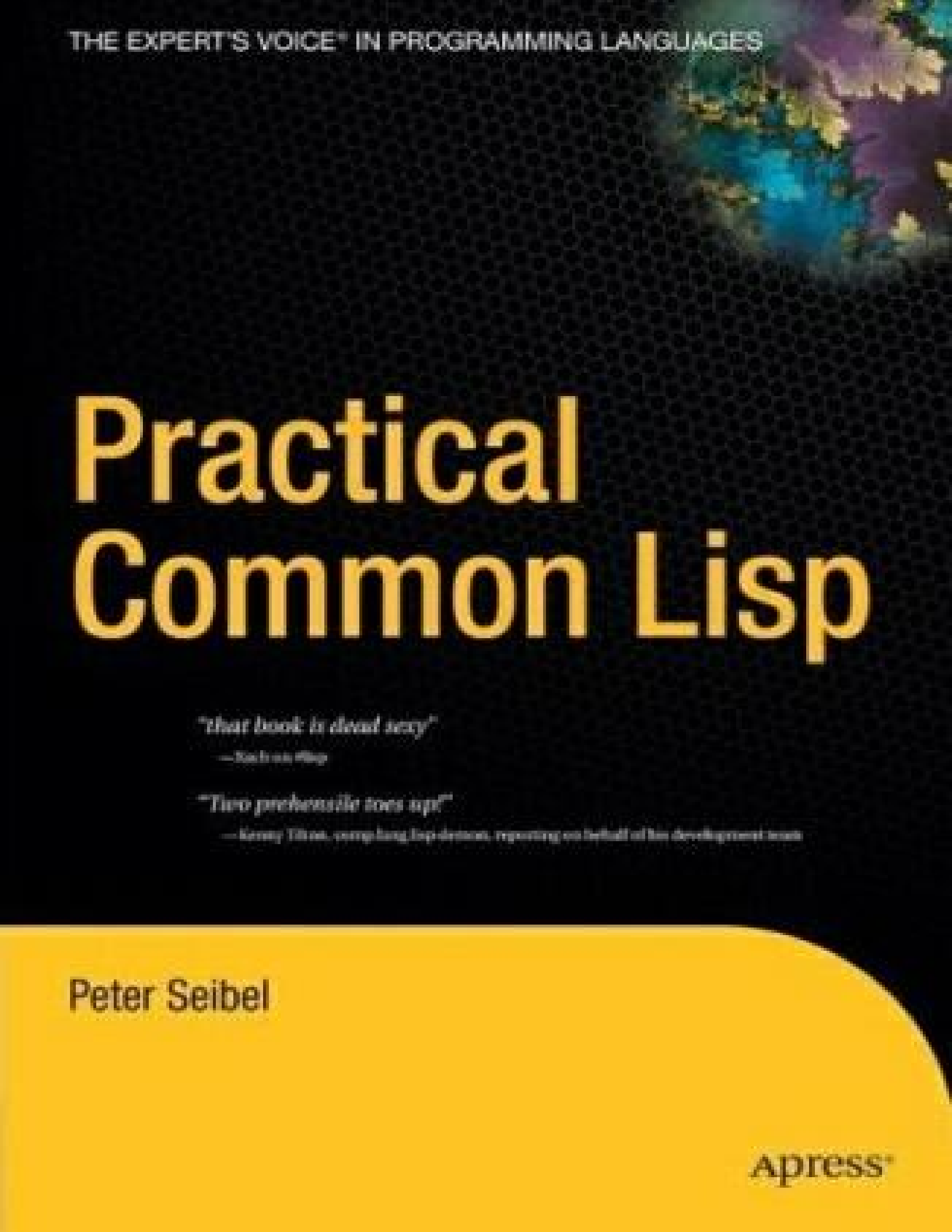

THE EXPERT'S VOICE® IN PROGRAMMING LANGUAGES
Practical Common Lisp
"that book is dead sexy"
"Two prehensile toes up!"
Peter Seibel
Apress®

that book is dead sexy —Xach on #lisp

(more blurbs)

This page, and the pages it links to, contain text of the Common Lisp book *Practical Common Lisp* published by Apress These pages now contain the final text as it appears in the book. If you find errors in these pages, please send email to book@gigamonkeys.com. These pages will remain online in perpetuity—I hope they will serve as a useful introduction to Common Lisp for folks who are curious about Lisp but maybe not yet curious enough to shell out big bucks for a dead-tree book and a good Common Lisp tutorial for folks who want to get down to real coding right away. However, don't let that stop you from buying the printed version available from Apress at your favorite local or online bookseller. For the complete bookstore browsing experience, you can read the letter to the reader that appears on the back cover of the treeware edition of the book.

1. Introduction: Why Lisp?
2. Lather, Rinse, Repeat: A Tour of the REPL
3. Practical: A Simple Database
4. Syntax and Semantics
5. Functions
6. Variables
7. Macros: Standard Control Constructs
8. Macros: Defining Your Own
9. Practical: Building a Unit Test Framework
10. Numbers, Characters, and Strings
11. Collections
12. They Called It LISP for a Reason: List Processing
13. Beyond Lists: Other Uses for Cons Cells
14. Files and File I/O
15. Practical: A Portable Pathname Library
16. Object Reorientation: Generic Functions
17. Object Reorientation: Classes
18. A Few FORMAT Recipes
19. Beyond Exception Handling: Conditions and Restarts
20. The Special Operators

21. Programming in the Large: Packages and Symbols
22. LOOP for Black Belts
23. Practical: A Spam Filter
24. Practical: Parsing Binary Files
25. Practical: An ID3 Parser
26. Practical: Web Programming with AllegroServe
27. Practical: An MP3 Database
28. Practical: A Shoutcast Server
29. Practical: An MP3 Browser
30. Practical: An HTML Generation Library, the Interpreter
31. Practical: An HTML Generation Library, the Compiler
32. Conclusion: What's Next?

Copyright © 2003-2009, Peter Seibel

“I have been complimented many times and they always embarrass me; I always feel that they have not said enough.” —Mark Twain

Blurbs

“that book is dead sexy”—**Xach on #lisp**

“Peter Seibel offers a fresh view of Lisp and its possibilities for elegantly solving problems. In Practical Common Lisp, he gives enough basic information to let you quickly see the power of the functional language paradigm. He then dazzles you with examples that seem almost magical in their simplicity and power. This read is pure fun from start to finish.” —**Gary Pollice, from Dr. Dobb's Portal, May 17, 2006 article on the 2006 Jolt Awards**

“Peter Seibel's *Practical Common Lisp* is just what the title implies: an excellent introduction to Common Lisp for someone who wants to dive in and start using the language for real work. The book is very well written and is fun to read—at least for those of us whose idea of fun extends to learning new programming languages.

Rather than spending a lot of time on abstract discussion of Lisp's place in the universe of programming lnaguages, Seibel dives right in, guiding the reader through a series of programming examples of increasing complexity. This approach places the most emphasis on those parts of Common Lisp that skilled programmers use the most, without getting bogged down in the odd corners of Common Lisp that even the experts must look up in the manual. The result of Seibel's example-driven approach is to give the reader an excellent appreciation of the power of Common Lisp in building complex, evolving software systems with a minimum of effort.

There are already many good books on Common Lisp that offer a more abstract and comparative approach, but a good ‘Here's how you do it—and why’ book, aimed at the working programmer, is a valuable contribution, both to current Common Lisp users and those who should be.” —**Scott E. Fahlman, Research Professor of Computer Science, Carnegie Mellon University**

“This book shows the power of Lisp not only in the areas that it has traditionally been noted for—such as developing a complete unit test framework in only 26 lines of code—but also in new areas such as parsing binary MP3 files, building a web application for browsing a collection of songs, and streaming audio over the web. Many readers will be surprised that Lisp allows you to do all this with conciseness similar to scripting languages such as Python, efficiency similar to C++, and unparalleled flexibility in designing your own language extensions.” —**Peter Norvig, Director of Search Quality, Google Inc; author of *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp***

“I wish this book had already existed when I started learning Lisp. It's not that there aren't other good books about (Common) Lisp out there, but none of them has such a pragmatic, up-to-date approach. And let's not forget that Peter covers topics like pathnames or conditions and restarts which are completely ignored in the rest of the Lisp literature.

If you're new to Lisp and want to dive right in don't hesitate to buy this book. Once you've read it and worked with it you can continue with the ‘classics’ like Graham, Norvig, Keene, or Steele.” —**Edi Weitz, maintainer of the *Common Lisp Cookbook* and author of CL-PPCRE regular expression library.**

“Two prehensile toes up!” —**Kenny Tilton, comp.lang.lisp demon, reporting on behalf of his development team.**

“Experienced programmers learn best from examples and it is delightful to see that Lisp is finally being served with Seibel's example-rich tutorial text. Especially delightful is the fact that this book includes so many examples that fall within the realm of problems today's programmers might be called upon to tackle, such as Web development and streaming media.—**Philip Greenspun, author of *Software Engineering for Internet Applications*, MIT Department of Electrical Engineering and Computer Science**

“*Practical Common Lisp* is an excellent book that covers the breadth of the Common Lisp language and also demonstrates the unique features of Common Lisp with real-world applications that the reader can run and extend. This book not only shows what Common Lisp is but also why every programmer should be familiar with Lisp.” —**John Foderaro, Senior Scientist, Franz Inc.**

“The Maxima Project frequently gets queries from potential new contributors who would like to learn Common Lisp. I am pleased to finally have a book that I can recommend to them without reservation. Peter Seibel's clear, direct style allows the reader to quickly appreciate the power of Common Lisp. His many included examples, which focus on contemporary programming problems, demonstrate that Lisp is much more than an academic programming language. *Practical Common Lisp* is a welcome addition to the literature.” —**James Amundson, Maxima Project Leader**

“I like the interspersed Practical chapters on 'real' and useful programs. We need books of this kind telling the world that crunching strings and numbers into trees or graphs is easily done in Lisp.—**Professor Christian Queinnec, Universite Paris 6 (Pierre et Marie Curie)**

“One of the most important parts of learning a programming language is learning its proper programming style. This is hard to teach, but it can be painlessly absorbed from *Practical Common Lisp*. Just reading the practical examples made me a better programmer in any language.” —**Peter Scott, Lisp programmer**

“Finally, a Lisp book for the rest of us. If you want to learn how to write a factorial function, this is not your book. Seibel writes for the practical programmer, emphasizing the engineer/artist over the scientist, subtly and gracefully implying the power of the language while solving understandable real-world problems.

In most chapters, the reading of the chapter feels just like the experience of writing a program, starting with a little understanding, then having that understanding grow, like building the shoulders upon which you can then stand. When Seibel introduced macros as an aside while building a test framework, I was shocked at how such a simple example made me really 'get' them. Narrative context is extremely powerful and the technical books that use it are a cut above. Congrats!” —**Keith Irwin, Lisp Programmer**

“While learning Lisp, one is often refered to the CL HyperSpec if they do not know what a particular function does, however, I found that I often did not 'get it' just reading the HyperSpec. When I had a problem of this manner, I turned to *Practical*

Common Lisp every single time—it is by far the most readable source on the subject that shows you how to program, not just tell you.” —**Philip Haddad, Lisp Programmer**

“With the IT world evolving at an ever increasing pace, professionals need the most powerful tools available. This is why Common Lisp—the most powerful, flexible, and stable programming language ever—is seeing such a rise in popularity. *Practical Common Lisp* is the long-awaited book that will help you harness the power of Common Lisp to tackle today's complex real world problems.” —**Marc Battyani, author of CL-PDF, CL-TYPESETTING, and mod_lisp.**

“Please don't assume Common Lisp is only useful for Databases, Unit Test Frameworks, Spam Filters, ID3 Parsers, Web Programming, Shoutcast Servers, HTML Generation Interpreters, and HTML Generation Compilers just because these are the only things happened to be implemented in the book *Practical Common Lisp*.—**Tobias C. Rittweiler, Lisp Programmer**

“When I met Peter, who just started writing this book, I asked to myself (not to him, of course) ‘why yet another book on Common Lisp, when there are many nice introductory books?’ One year later, I found a draft of the new book and recognized I was wrong. This book is not ‘yet another’ one. The author focuses on practical aspects rather than on technical details of the language. When I first studied Lisp by reading an introductory book, I felt I understood the language, but I also had an impression ‘so what?’, meaning I had no idea about how to use it. In contrast, this book leaps into a ‘PRACTICAL’ chapter after the first few chapters that explain the very basic notions of the language. Then the readers are expected to learn more about the language while they are following the

PRACTICAL projects, which are combined together to form a product of a significant size. After reading this book, the readers will feel themselves expert programmers on Common Lisp since they have ‘finished’ a big project already. I think Lisp is the only language that allows this type of practical introduction. Peter makes use of this feature of the language in building up a fancy introduction on Common Lisp.” —**Taiichi Yuasa, Professor, Department of Communications and Computer Engineering, Kyoto University**

Have something to say about this book? Something nice? Want to see it here? Send it along to book@gigamonkeys.com.

Back to index.

Copyright © 2003-2004, Peter Seibel

Dear Reader,

Practical Common Lisp ... isn't that an oxymoron? If you're like most programmers, you probably know something about Lisp—from a comp sci course in college or from learning enough Elisp to customize Emacs a bit. Or maybe you just know someone who won't shut up about Lisp, the greatest language ever. But you probably never figured you'd see *practical* and *Lisp* in the same book title.

Yet, you're reading this; you must want to know more. Maybe you believe learning Lisp will make you a better programmer in any language. Or maybe you just want to know what those Lisp fanatics are yammering about all the time. Or maybe you have learned some Lisp but haven't quite made the leap to using it to write interesting software.

If any of those is true, this book is for you. Using Common Lisp, an ANSI standardized, industrial-strength dialect of Lisp, I show you how to write software that goes well beyond silly academic exercises or trivial editor customizations. And I show you how Lisp—even with many of its features adopted by other languages—still has a few tricks up its sleeve.

But unlike many Lisp books, this one doesn't just touch on a few of Lisp's greatest features and then leave you on your own to actually use them. I cover all the language features you'll need to write real programs and devote well over a third of the book to developing nontrivial software—a statistical spam filter, a library for parsing binary files, and

a server for streaming MP3s over a network complete with an online MP3 database and Web interface.

So turn the book over, open it up, and see for yourself how eminently practical using the greatest language ever invented can be.

Sincerely,

Peter Seibel

Back to index.

Copyright © 2003-2004, Peter Seibel

1. Introduction: Why Lisp?

If you think the greatest pleasure in programming comes from getting a lot done with code that simply and clearly expresses your intention, then programming in Common Lisp is likely to be about the most fun you can have with a computer. You'll get more done, faster, using it than you would using pretty much any other language.

That's a bold claim. Can I justify it? Not in a just a few pages in this chapter--you're going to have to learn some Lisp and see for yourself--thus the rest of this book. For now, let me start with some anecdotal evidence, the story of my own road to Lisp. Then, in the next section, I'll explain the payoff I think you'll get from learning Common Lisp.

I'm one of what must be a fairly small number of second-generation Lisp hackers. My father got his start in computers writing an operating system in assembly for the machine he used to gather data for his doctoral dissertation in physics. After running computer systems at various physics labs, by the 1980s he had left physics altogether and was working at a large pharmaceutical company. That company had a project under way to develop software to model production processes in its chemical plants--if you increase the size of this vessel, how does it affect annual production? The original team, writing in FORTRAN, had burned through half the money and almost all the time allotted to the project with nothing to show for their efforts. This being the 1980s and the middle of the artificial intelligence (AI) boom, Lisp was in the air. So my dad--at that point

not a Lisper--went to Carnegie Mellon University (CMU) to talk to some of the folks working on what was to become Common Lisp about whether Lisp might be a good language for this project.

The CMU folks showed him some demos of stuff they were working on, and he was convinced. He in turn convinced his bosses to let his team take over the failing project and do it in Lisp. A year later, and using only what was left of the original budget, his team delivered a working application with features that the original team had given up any hope of delivering. My dad credits his team's success to their decision to use Lisp.

Now, that's just one anecdote. And maybe my dad is wrong about why they succeeded. Or maybe Lisp was better only in comparison to other languages of the day. These days we have lots of fancy new languages, many of which have incorporated features from Lisp. Am I really saying Lisp can offer you the same benefits today as it offered my dad in the 1980s? Read on.

Despite my father's best efforts, I didn't learn any Lisp in high school. After a college career that didn't involve much programming in any language, I was seduced by the Web and back into computers. I worked first in Perl, learning enough to be dangerous while building an online discussion forum for *Mother Jones* magazine's Web site and then moving to a Web shop, Organic Online, where I worked on big--for the time--Web sites such as the one Nike put up during the 1996 Olympics. Later I moved onto Java as an early developer at WebLogic, now part of BEA. After WebLogic, I joined another startup where I was the lead programmer on a team building a transactional

messaging system in Java. Along the way, my general interest in programming languages led me to explore such mainstream languages as C, C++, and Python, as well as less well-known ones such as Smalltalk, Eiffel, and Beta.

So I knew two languages inside and out and was familiar with another half dozen. Eventually, however, I realized my interest in programming languages was really rooted in the idea planted by my father's tales of Lisp--that different languages really are different, and that, despite the formal Turing equivalence of all programming languages, you really can get more done more quickly in some languages than others and have more fun doing it. Yet, ironically, I had never spent that much time with Lisp itself. So, I started doing some Lisp hacking in my free time. And whenever I did, it was exhilarating how quickly I was able to go from idea to working code.

For example, one vacation, having a week or so to hack Lisp, I decided to try writing a version of a program--a system for breeding genetic algorithms to play the game of Go--that I had written early in my career as a Java programmer. Even handicapped by my then rudimentary knowledge of Common Lisp and having to look up even basic functions, it still felt more productive than it would have been to rewrite the same program in Java, even with several extra years of Java experience acquired since writing the first version.

A similar experiment led to the library I'll discuss in Chapter 24. Early in my time at WebLogic I had written a library, in Java, for taking apart Java class files. It worked, but the code was a bit of a mess and hard to modify or extend. I had tried several times, over the

years, to rewrite that library, thinking that with my ever-improving Java chops I'd find some way to do it that didn't bog down in piles of duplicated code. I never found a way. But when I tried to do it in Common Lisp, it took me only two days, and I ended up not only with a Java class file parser but with a general-purpose library for taking apart any kind of binary file. You'll see how that library works in Chapter 24 and use it in Chapter 25 to write a parser for the ID3 tags embedded in MP3 files.

Why Lisp?

It's hard, in only a few pages of an introductory chapter, to explain why users of a language like it, and it's even harder to make the case for why you should invest your time in learning a certain language. Personal history only gets us so far. Perhaps I like Lisp because of some quirk in the way my brain is wired. It could even be genetic, since my dad has it too. So before you dive into learning Lisp, it's reasonable to want to know what the payoff is going to be.

For some languages, the payoff is relatively obvious. For instance, if you want to write low-level code on Unix, you should learn C. Or if you want to write certain kinds of cross-platform applications, you should learn Java. And any of a number companies still use a lot of C++, so if you want to get a job at one of them, you should learn C++.

For most languages, however, the payoff isn't so easily categorized; it has to do with subjective criteria such as how it feels to use the language. Perl advocates like to say that Perl "makes easy things easy and hard things possible" and revel in the fact that, as the Perl motto has it, "There's more than one way to do it."[1] Python's fans, on the other hand, think Python is clean and simple and think Python code is easier to understand because, as *their* motto says, "There's only one way to do it."

So, why Common Lisp? There's no immediately obvious payoff for adopting Common Lisp the way there is for C, Java, and C++ (unless, of course, you happen to own a Lisp Machine). The benefits

of using Lisp have much more to do with the experience of using it. I'll spend the rest of this book showing you the specific features of Common Lisp and how to use them so you can see for yourself what it's like. For now I'll try to give you a sense of Lisp's philosophy.

The nearest thing Common Lisp has to a motto is the koan-like description, "the programmable programming language." While cryptic, that description gets at the root of the biggest advantage Common Lisp still has over other languages. More than any other language, Common Lisp follows the philosophy that what's good for the language's designer is good for the language's users. Thus, when you're programming in Common Lisp, you almost never find yourself wishing the language supported some feature that would make your program easier to write, because, as you'll see throughout this book, you can just add the feature yourself.

Consequently, a Common Lisp program tends to provide a much clearer mapping between your ideas about how the program works and the code you actually write. Your ideas aren't obscured by boilerplate code and endlessly repeated idioms. This makes your code easier to maintain because you don't have to wade through reams of code every time you need to make a change. Even systemic changes to a program's behavior can often be achieved with relatively small changes to the actual code. This also means you'll develop code more quickly; there's less code to write, and you don't waste time thrashing around trying to find a clean way to express yourself within the limitations of the language.[2]

Common Lisp is also an excellent language for exploratory programming--if you don't know exactly how your program is going to work when you first sit down to write it, Common Lisp provides several features to help you develop your code incrementally and interactively.

For starters, the interactive read-eval-print loop, which I'll introduce in the next chapter, lets you continually interact with your program as you develop it. Write a new function. Test it. Change it. Try a different approach. You never have to stop for a lengthy compilation cycle.[3]

Other features that support a flowing, interactive programming style are Lisp's dynamic typing and the Common Lisp condition system. Because of the former, you spend less time convincing the compiler you should be allowed to run your code and more time actually running it and working on it,[4] and the latter lets you develop even your error handling code interactively.

Another consequence of being "a programmable programming language" is that Common Lisp, in addition to incorporating small changes that make particular programs easier to write, can easily adopt big new ideas about how programming languages should work. For instance, the original implementation of the Common Lisp Object System (CLOS), Common Lisp's powerful object system, was as a library written in portable Common Lisp. This allowed Lisp programmers to gain actual experience with the facilities it provided before it was officially incorporated into the language.

Whatever new paradigm comes down the pike next, it's extremely likely that Common Lisp will be able to absorb it without requiring

any changes to the core language. For example, a Lisper has recently written a library, AspectL, that adds support for aspect-oriented programming (AOP) to Common Lisp.[5] If AOP turns out to be the next big thing, Common Lisp will be able to support it without any changes to the base language and without extra preprocessors and extra compilers.[6]

Where It Began

Common Lisp is the modern descendant of the Lisp language first conceived by John McCarthy in 1956. Lisp circa 1956 was designed for "symbolic data processing"[7] and derived its name from one of the things it was quite good at: LISt Processing. We've come a long way since then: Common Lisp sports as fine an array of modern data types as you can ask for: a condition system that, as you'll see in Chapter 19, provides a whole level of flexibility missing from the exception systems of languages such as Java, Python, and C++; powerful facilities for doing object-oriented programming; and several language facilities that just don't exist in other programming languages. How is this possible? What on Earth would provoke the evolution of such a well-equipped language?

Well, McCarthy was (and still is) an artificial intelligence (AI) researcher, and many of the features he built into his initial version of the language made it an excellent language for AI programming. During the AI boom of the 1980s, Lisp remained a favorite tool for programmers writing software to solve hard problems such as automated theorem proving, planning and scheduling, and computer vision. These were problems that required a lot of hard-to-write software; to make a dent in them, AI programmers needed a powerful language, and they grew Lisp into the language they needed. And the Cold War helped--as the Pentagon poured money into the Defense Advanced Research Projects Agency (DARPA), a lot of it went to folks working on problems such as large-scale battlefield simulations, automated planning, and natural language

interfaces. These folks also used Lisp and continued pushing it to do what they needed.

The same forces that drove Lisp's feature evolution also pushed the envelope along other dimensions--big AI problems eat up a lot of computing resources however you code them, and if you run Moore's law in reverse for 20 years, you can imagine how scarce computing resources were on circa-80s hardware. The Lisp guys had to find all kinds of ways to squeeze performance out of their implementations. Modern Common Lisp implementations are the heirs to those early efforts and often include quite sophisticated, native machine code-generating compilers. While today, thanks to Moore's law, it's possible to get usable performance from a purely interpreted language, that's no longer an issue for Common Lisp. As I'll show in Chapter 32, with proper (optional) declarations, a good Lisp compiler can generate machine code quite similar to what might be generated by a C compiler.

The 1980s were also the era of the Lisp Machines, with several companies, most famously Symbolics, producing computers that ran Lisp natively from the chips up. Thus, Lisp became a systems programming language, used for writing the operating system, editors, compilers, and pretty much everything else that ran on the Lisp Machines.

In fact, by the early 1980s, with various AI labs and the Lisp machine vendors all providing their own Lisp implementations, there was such a proliferation of Lisp systems and dialects that the folks at DARPA began to express concern about the Lisp community splintering. To

address this concern, a grassroots group of Lisp hackers got together in 1981 and began the process of standardizing a new language called Common Lisp that combined the best features from the existing Lisp dialects. Their work was documented in the book *Common Lisp the Language* by Guy Steele (Digital Press, 1984)--*CLtL* to the Lisp-cognoscenti.

By 1986 the first Common Lisp implementations were available, and the writing was on the wall for the dialects it was intended to replace. In 1996, the American National Standards Institute (ANSI) released a standard for Common Lisp that built on and extended the language specified in CLtL, adding some major new features such as the CLOS and the condition system. And even that wasn't the last word: like CLtL before it, the ANSI standard intentionally leaves room for implementers to experiment with the best way to do things: a full Lisp implementation provides a rich runtime environment with access to GUI widgets, multiple threads of control, TCP/IP sockets, and more. These days Common Lisp is evolving much like other open-source languages--the folks who use it write the libraries they need and often make them available to others. In the last few years, in particular, there has been a spurt of activity in open-source Lisp libraries.

So, on one hand, Lisp is one of computer science's "classical" languages, based on ideas that have stood the test of time.[8] On the other, it's a thoroughly modern, general-purpose language whose design reflects a deeply pragmatic approach to solving real problems as efficiently and robustly as possible. The only downside of Lisp's "classical" heritage is that lots of folks are still walking around with

ideas about Lisp based on some particular flavor of Lisp they were exposed to at some particular time in the nearly half century since McCarthy invented Lisp. If someone tells you Lisp is only interpreted, that it's slow, or that you have to use recursion for everything, ask them what dialect of Lisp they're talking about and whether people were wearing bell-bottoms when they learned it.[9]

But I learned Lisp Once, And IT Wasn't Like what you're describing

If you've used Lisp in the past, you may have ideas about what "Lisp" is that have little to do with Common Lisp. While Common Lisp supplanted most of the dialects it's descended from, it isn't the only remaining Lisp dialect, and depending on where and when you were exposed to Lisp, you may very well have learned one of these other dialects.

Other than Common Lisp, the one general-purpose Lisp dialect that still has an active user community is Scheme. Common Lisp borrowed a few important features from Scheme but never intended to replace it.

Originally designed at M.I.T., where it was quickly put to use as a teaching language for undergraduate computer science courses, Scheme has always been aimed at a different language niche than Common Lisp. In particular, Scheme's designers have focused on keeping the core language as small and as simple as possible. This has obvious benefits for a teaching language and also for programming language researchers who like to be able to formally prove things about languages.

It also has the benefit of making it relatively easy to understand the whole language as specified in the standard. But, it does so at the cost of omitting many useful features that are standardized in Common Lisp. Individual Scheme implementations may provide these features in implementation-specific ways, but their omission from the standard makes it harder to write portable Scheme code than to write portable Common Lisp code.

Scheme also emphasizes a functional programming style and the use of recursion much more than Common Lisp does. If you studied Lisp in college and came away with the impression that it was only an academic language with no real-world application, chances are you learned Scheme. This isn't to say that's a particularly fair characterization of Scheme, but it's even less applicable to Common Lisp, which was expressly designed to be a real-world engineering language rather than a theoretically "pure" language.

If you've learned Scheme, you should also be aware that a number of subtle differences between Scheme and Common Lisp may trip you up. These differences are also the basis for several perennial religious wars between the hotheads in the Common Lisp and Scheme communities. I'll try to point out some of the more important differences as we go along.

Two other Lisp dialects still in widespread use are Elisp, the extension language for the Emacs editor, and Autolisp, the extension language for Autodesk's AutoCAD computer-aided design tool. Although it's possible more lines of Elisp and Autolisp have been written than of any other dialect of Lisp, neither can be used outside their host application, and both are quite old-fashioned Lisps compared to either Scheme or Common Lisp. If you've used one of these dialects, prepare to hop in the Lisp time machine and jump forward several decades.

Who This Book Is For

This book is for you if you're curious about Common Lisp, regardless of whether you're already convinced you want to use it or if you just want to know what all the fuss is about.

If you've learned some Lisp already but have had trouble making the leap from academic exercises to real programs, this book should get you on your way. On the other hand, you don't have to be already convinced that you want to use Lisp to get something out of this book.

If you're a hard-nosed pragmatist who wants to know what advantages Common Lisp has over languages such as Perl, Python, Java, C, or C#, this book should give you some ideas. Or maybe you don't even care about using Lisp--maybe you're already sure Lisp isn't really any better than other languages you know but are annoyed by some Lisper telling you that's because you just don't "get it." If so, this book will give you a straight-to-the-point introduction to Common Lisp. If, after reading this book, you still think Common Lisp is no better than your current favorite languages, you'll be in an excellent position to explain exactly why.

I cover not only the syntax and semantics of the language but also how you can use it to write software that does useful stuff. In the first part of the book, I'll cover the language itself, mixing in a few "practical" chapters, where I'll show you how to write real code. Then, after I've covered most of the language, including several parts that other books leave for you to figure out on your own, the

remainder of the book consists of nine more practical chapters where I'll help you write several medium-sized programs that actually do things you might find useful: filter spam, parse binary files, catalog MP3s, stream MP3s over a network, and provide a Web interface for the MP3 catalog and server.

After you finish this book, you'll be familiar with all the most important features of the language and how they fit together, you'll have used Common Lisp to write several nontrivial programs, and you'll be well prepared to continue exploring the language on your own. While everyone's road to Lisp is different, I hope this book will help smooth the way for you. So, let's begin.

---

[1]Perl is also worth learning as "the duct tape of the Internet."

[2]Unfortunately, there's little actual research on the productivity of different languages. One report that shows Lisp coming out well compared to C++ and Java in the combination of programmer and program efficiency is discussed at `http://www.norvig.com/java-lisp.html`.

[3]Psychologists have identified a state of mind called *flow* in which we're capable of incredible concentration and productivity. The importance of flow to programming has been recognized for nearly two decades since it was discussed in the classic book about human factors in programming *Peopleware: Productive Projects and Teams* by Tom DeMarco and Timothy Lister (Dorset House, 1987). The two key facts about flow are that it takes around 15 minutes to get into a state of flow and that even brief interruptions can break you right out of it, requiring another 15-minute immersion to reenter. DeMarco and Lister, like most subsequent authors, concerned themselves mostly with flow-destroying interruptions such as ringing telephones and inopportune visits from the boss. Less frequently considered but probably just as important to programmers are the interruptions caused by our tools. Languages that require, for instance, a lengthy compilation before you can try your latest code can be just as inimical to flow as a noisy phone or a nosy boss. So, one way to look at Lisp is as a language designed to keep you in a state of flow.

[4]This point is bound to be somewhat controversial, at least with some folks. Static versus dynamic typing is one of the classic religious wars in programming. If you're coming from C++ and Java (or from statically typed functional languages such as Haskel and ML) and refuse to consider living without static type checks, you might as well put this book down now. However, before you do, you might first want to check out what self-described "statically typed bigot" Robert Martin (author of *Designing Object Oriented C++ Applications Using the Booch Method* [Prentice Hall, 1995]) and C++ and Java author Bruce Eckel (author of *Thinking in C++* [Prentice Hall, 1995] and *Thinking in Java* [Prentice Hall, 1998]) have had to say about dynamic typing on their weblogs (`http://www.artima.com/weblogs/viewpost.jsp?thread=4639` and `http://www.mindview.net/WebLog/log-0025`). On the other hand, folks coming from Smalltalk, Python, Perl, or Ruby should feel right at home with this aspect of Common Lisp.

[5]AspectL is an interesting project insofar as AspectJ, its Java-based predecessor, was written by Gregor Kiczales, one of the designers of Common Lisp's object and metaobject systems. To many Lispers, AspectJ seems like Kiczales's attempt to backport his ideas from Common Lisp into Java. However, Pascal Costanza, the author of AspectL, thinks there are interesting ideas in AOP that could be useful in Common Lisp. Of course, the reason he's able to implement AspectL as a library is because of the incredible flexibility of the Common Lisp Meta Object Protocol Kiczales designed. To implement AspectJ, Kiczales had to write what was essentially a separate compiler that compiles a new language into Java source code. The AspectL project page is at `http://common-lisp.net/ project/aspectl/`.

[6]Or to look at it another, more technically accurate, way, Common Lisp comes with a built-in facility for integrating compilers for embedded languages.

[7]*Lisp 1.5 Programmer's Manual* (M.I.T. Press, 1962)

[8]Ideas first introduced in Lisp include the if/then/else construct, recursive function calls, dynamic memory allocation, garbage collection, first-class functions, lexical closures, interactive programming, incremental compilation, and dynamic typing.

[9]One of the most commonly repeated myths about Lisp is that it's "dead." While it's true that Common Lisp isn't as widely used as, say, Visual Basic or Java, it seems strange to describe a language that continues to be used for new development and that continues to attract new users as "dead." Some recent Lisp success stories include Paul Graham's Viaweb, which became Yahoo Store when Yahoo bought his company; ITA Software's airfare pricing and shopping system, QPX, used by the online ticket seller Orbitz and others; Naughty Dog's game for the PlayStation 2, Jak and Daxter, which is largely written in a

domain-specific Lisp dialect Naughty Dog invented called GOAL, whose compiler is itself written in Common Lisp; and the Roomba, the autonomous robotic vacuum cleaner, whose software is written in L, a downwardly compatible subset of Common Lisp. Perhaps even more telling is the growth of the Common-Lisp.net Web site, which hosts open-source Common Lisp projects, and the number of local Lisp user groups that have sprung up in the past couple of years.

Copyright © 2003-2005, Peter Seibel

2. Lather, Rinse, Repeat: A Tour of the REPL

In this chapter you'll set up your programming environment and write your first Common Lisp programs. We'll use the easy-to-install Lisp in a Box developed by Matthew Danish and Mikel Evins, which packages a Common Lisp implementation with Emacs, a powerful Lisp-aware text editor, and SLIME,[1] a Common Lisp development environment built on top of Emacs.

This combo provides a state-of-the-art Common Lisp development environment that supports the incremental, interactive development style that characterizes Lisp programming. The SLIME environment has the added advantage of providing a fairly uniform user interface regardless of the operating system and Common Lisp implementation you choose. I'll use the Lisp in a Box environment in order to have a specific development environment to talk about; folks who want to explore other development environments such as the graphical integrated development environments (IDEs) provided by some of the commercial Lisp vendors or environments based on other editors shouldn't have too much trouble translating the basics.[2]

Choosing a Lisp Implementation

The first thing you have to do is to choose a Lisp implementation. This may seem like a strange thing to have to do for folks used to languages such as Perl, Python, Visual Basic (VB), C#, and Java. The difference between Common Lisp and these languages is that Common Lisp is defined by its standard--there is neither a single implementation controlled by a benevolent dictator, as with Perl and Python, nor a canonical implementation controlled by a single company, as with VB, C#, and Java. Anyone who wants to read the standard and implement the language is free to do so. Furthermore, changes to the standard have to be made in accordance with a process controlled by the standards body American National Standards Institute (ANSI). That process is designed to keep any one entity, such as a single vendor, from being able to arbitrarily change the standard.[3] Thus, the Common Lisp standard is a contract between any Common Lisp vendor and Common Lisp programmers. The contract tells you that if you write a program that uses the features of the language the way they're described in the standard, you can count on your program behaving the same in any conforming implementation.

On the other hand, the standard may not cover everything you may want to do in your programs--some things were intentionally left unspecified in order to allow continuing experimentation by implementers in areas where there wasn't consensus about the best way for the language to support certain features. So every implementation offers some features above and beyond what's

specified in the standard. Depending on what kind of programming you're going to be doing, it may make sense to just pick one implementation that has the extra features you need and use that. On the other hand, if we're delivering Lisp source to be used by others, such as libraries, you'll want--as far as possible--to write portable Common Lisp. For writing code that should be mostly portable but that needs facilities not defined by the standard, Common Lisp provides a flexible way to write code "conditionalized" on the features available in a particular implementation. You'll see an example of this kind of code in Chapter 15 when we develop a simple library that smoothes over some differences between how different Lisp implementations deal with filenames.

For the moment, however, the most important characteristic of an implementation is whether it runs on our favorite operating system. The folks at Franz, makers of Allegro Common Lisp, are making available a trial version of their product for use with this book that runs on Linux, Windows, and OS X. Folks looking for an open-source implementation have several options. SBCL[4] is a high-quality open-source implementation that compiles to native code and runs on a wide variety of Unixes, including Linux and OS X. SBCL is derived from CMUCL,[5] which is a Common Lisp developed at Carnegie Mellon University, and, like CMUCL, is largely in the public domain, except a few sections licensed under Berkeley Software Distribution (BSD) style licenses. CMUCL itself is another fine choice, though SBCL tends to be easier to install and now supports 21-bit Unicode.[6] For OS X users, OpenMCL is an excellent choice--it compiles to machine code, supports threads, and has quite good integration with OS X's Carbon and Cocoa toolkits. Other open-

source and commercial implementations are available. See Chapter 32 for resources from which you can get more information.

All the Lisp code in this book should work in any conforming Common Lisp implementation unless otherwise noted, and SLIME will smooth out some of the differences between implementations by providing us with a common interface for interacting with Lisp. The output shown in this book is from Allegro running on GNU/Linux; in some cases, other Lisp's may generate slightly different error messages or debugger output.

Getting Up and Running with Lisp in a Box

Since the Lisp in a Box packaging is designed to get new Lispers up and running in a first-rate Lisp development environment with minimum hassle, all you need to do to get it running is to grab the appropriate package for your operating system and the preferred Lisp from the Lisp in a Box Web site listed in Chapter 32 and then follow the installation instructions.

Since Lisp in a Box uses Emacs as its editor, you'll need to know at least a bit about how to use it. Perhaps the best way to get started with Emacs is to work through its built-in tutorial. To start the tutorial, select the first item of the Help menu, Emacs tutorial. Or press the Ctrl key, type `h`, release the Ctrl key, and then press `t`. Most Emacs commands are accessible via such key combinations; because key combinations are so common, Emacs users have a notation for describing key combinations that avoids having to constantly write out combinations such as "Press the Ctrl key, type `h`, release the Ctrl key, and then press `t`." Keys to be pressed together--a so-called key chord--are written together and separated by a hyphen. Keys, or key chords, to be pressed in sequence are separated by spaces. In a key chord, `C` represents the Ctrl key and `M` represents the Meta key (also known as Alt). Thus, we could write the key combination we just described that starts the tutorial like so: `C-h t`.

The tutorial describes other useful commands and the key combinations that invoke them. Emacs also comes with extensive online documentation using its own built-in hypertext documentation browser, Info. To read the manual, type `C-h i`. The Info system

comes with its own tutorial, accessible simply by pressing `h` while reading the manual. Finally, Emacs provides quite a few ways to get help, all bound to key combos starting with `C-h`. Typing `C-h ?` brings up a complete list. Two of the most useful, besides the tutorial, are `C-h k`, which lets us type any key combo and tells us what command it invokes, and `C-h w`, which lets us enter the name of a command and tells us what key combination invokes it.

The other crucial bit of Emacs terminology, for folks who refuse to work through the tutorial, is the notion of a *buffer*. While working in Emacs, each file you edit will be represented by a different buffer, only one of which is "current" at any given time. The current buffer receives all input--whatever you type and any commands you invoke. Buffers are also used to represent interactions with programs such as Common Lisp. Thus, one common action you'll take is to "switch buffers," which means to make a different buffer the current buffer so you can edit a particular file or interact with a particular program. The command `switch-to-buffer`, bound to the key combination `C-x b`, prompts for the name of a buffer in the area at the bottom of the Emacs frame. When entering a buffer name, hitting Tab will complete the name based on the characters typed so far or will show a list of possible completions. The prompt also suggests a default buffer, which you can accept just by hitting Return. You can also switch buffers by selecting a buffer from the Buffers menu.

In certain contexts, other key combinations may be available for switching to certain buffers. For instance, when editing Lisp source files, the key combo `C-c C-z` switches to the buffer where you interact with Lisp.

Free Your Mind: Interactive Programming

When you start Lisp in a Box, you should see a buffer containing a prompt that looks like this:

```
CL-USER>
```

This is the Lisp prompt. Like a Unix or DOS shell prompt, the Lisp prompt is a place where you can type expressions that will cause things to happen. However, instead of reading and interpreting a line of shell commands, Lisp reads Lisp expressions, evaluates them according to the rules of Lisp, and prints the result. Then it does it again with the next expression you type. That endless cycle of reading, evaluating, and printing is why it's called the *read-eval-print loop*, or REPL for short. It's also referred to as the *top-level*, the *top-level listener*, or the *Lisp listener*.

From within the environment provided by the REPL, you can define and redefine program elements such as variables, functions, classes, and methods; evaluate any Lisp expression; load files containing Lisp source code or compiled code; compile whole files or individual functions; enter the debugger; step through code; and inspect the state of individual Lisp objects.

All those facilities are built into the language, accessible via functions defined in the language standard. If you had to, you could build a pretty reasonable programming environment out of just the REPL and any text editor that knows how to properly indent Lisp code. But for the true Lisp programming experience, you need an environment, such as SLIME, that lets you interact with Lisp both via the REPL

and while editing source files. For instance, you don't want to have to cut and paste a function definition from a source file to the REPL or have to load a whole file just because you changed one function; your Lisp environment should let us evaluate or compile both individual expressions and whole files directly from your editor.

Experimenting in the REPL

To try the REPL, you need a Lisp expression that can be read, evaluated, and printed. One of the simplest kinds of Lisp expressions is a number. At the Lisp prompt, you can type `10` followed by Return and should see something like this:

```
CL-USER> 10
10
```

The first `10` is the one you typed. The Lisp reader, the *R* in REPL, reads the text "10" and creates a Lisp object representing the number 10. This object is a *self-evaluating* object, which means that when given to the evaluator, the *E* in REPL, it evaluates to itself. This value is then given to the printer, which prints the `10` on the line by itself. While that may seem like a lot of work just to get back to where you started, things get a bit more interesting when you give Lisp something meatier to chew on. For instance, you can type `(+ 2 3)` at the Lisp prompt.

```
CL-USER> (+ 2 3)
5
```

Anything in parentheses is a list, in this case a list of three elements, the symbol `+`, and the numbers 2 and 3. Lisp, in general, evaluates lists by treating the first element as the name of a function and the rest of the elements as expressions to be evaluated to yield the arguments to the function. In this case, the symbol `+` names a function that performs addition. 2 and 3 evaluate to themselves and are then passed to the addition function, which returns 5. The value 5 is passed to the printer, which prints it. Lisp can evaluate a list

expression in other ways, but we needn't get into them right away. First we have to write. . .

"Hello, World," Lisp Style

No programming book is complete without a "hello, world"[7] program. As it turns out, it's trivially easy to get the REPL to print "hello, world."

```
CL-USER> "hello, world"
"hello, world"
```

This works because strings, like numbers, have a literal syntax that's understood by the Lisp reader and are self-evaluating objects: Lisp reads the double-quoted string and instantiates a string object in memory that, when evaluated, evaluates to itself and is then printed in the same literal syntax. The quotation marks aren't part of the string object in memory--they're just the syntax that tells the reader to read a string. The printer puts them back on when it prints the string because it tries to print objects in the same syntax the reader understands.

However, this may not really qualify as a "hello, world" *program*. It's more like the "hello, world" *value*.

You can take a step toward a real program by writing some code that as a side effect prints the string "hello, world" to standard output. Common Lisp provides a couple ways to emit output, but the most flexible is the **FORMAT** function. **FORMAT** takes a variable number of arguments, but the only two required arguments are the place to send the output and a string. You'll see in the next chapter how the string can contain embedded directives that allow you to interpolate subsequent arguments into the string, ŕ la `printf` or Python's

string-`%`. As long as the string doesn't contain an `~`, it will be emitted as-is. If you pass `t` as its first argument, it sends its output to standard output. So a **FORMAT** expression that will print "hello, world" looks like this:[8]

```
CL-USER> (format t "hello, world")
hello, world
NIL
```

One thing to note about the result of the **FORMAT** expression is the **NIL** on the line after the "hello, world" output. That **NIL** is the result of evaluating the **FORMAT** expression, printed by the REPL. (**NIL** is Lisp's version of false and/or null. More on that in Chapter 4.) Unlike the other expressions we've seen so far, a **FORMAT** expression is more interesting for its side effect--printing to standard output in this case--than for its return value. But every expression in Lisp evaluates to some result.[9]

However, it's still arguable whether you've yet written a true "program." But you're getting there. And you're seeing the bottom-up style of programming supported by the REPL: you can experiment with different approaches and build a solution from parts you've already tested. Now that you have a simple expression that does what you want, you just need to package it in a function. Functions are one of the basic program building blocks in Lisp and can be defined with a **DEFUN** expression such as this:

```
CL-USER> (defun hello-world () (format t "hello, world"))
HELLO-WORLD
```

The `hello-world` after the **DEFUN** is the name of the function. In Chapter 4 we'll look at exactly what characters can be used in a name, but for now suffice it to say that lots of characters, such as `-`,

that are illegal in names in other languages are legal in Common Lisp. It's standard Lisp style--not to mention more in line with normal English typography--to form compound names with hyphens, such as `hello-world`, rather than with underscores, as in `hello_world`, or with inner caps such as `helloWorld`. The `()`s after the name delimit the parameter list, which is empty in this case because the function takes no arguments. The rest is the body of the function.

At one level, this expression, like all the others you've seen, is just another expression to be read, evaluated, and printed by the REPL. The return value in this case is the name of the function you just defined.[10] But like the **FORMAT** expression, this expression is more interesting for the side effects it has than for its return value. Unlike the **FORMAT** expression, however, the side effects are invisible: when this expression is evaluated, a new function that takes no arguments and with the body `(format t "hello, world")` is created and given the name `HELLO-WORLD`.

Once you've defined the function, you can call it like this:

```
CL-USER> (hello-world)
hello, world
NIL
```

You can see that the output is just the same as when you evaluated the **FORMAT** expression directly, including the **NIL** value printed by the REPL. Functions in Common Lisp automatically return the value of the last expression evaluated.

Saving Your Work

You could argue that this is a complete "hello, world" program of sorts. However, it still has a problem. If you exit Lisp and restart, the function definition will be gone. Having written such a fine function, you'll want to save your work.

Easy enough. You just need to create a file in which to save the definition. In Emacs you can create a new file by typing `C-x C-f` and then, when Emacs prompts you, entering the name of the file you want to create. It doesn't matter particularly where you put the file. It's customary to name Common Lisp source files with a `.lisp` extension, though some folks use `.cl` instead.

Once you've created the file, you can type the definition you previously entered at the REPL. Some things to note are that after you type the opening parenthesis and the word **DEFUN**, at the bottom of the Emacs window, SLIME will tell you the arguments expected. The exact form will depend somewhat on what Common Lisp implementation you're using, but it'll probably look something like this:

```
(defun name varlist &rest body)
```

The message will disappear as you start to type each new element but will reappear each time you enter a space. When you're entering the definition in the file, you might choose to break the definition across two lines after the parameter list. If you hit Return and then Tab, SLIME will automatically indent the second line appropriately, like this:[11]

```
(defun hello-world ()
  (format t "hello, world"))
```

SLIME will also help match up the parentheses--as you type a closing parenthesis, it will flash the corresponding opening parenthesis. Or you can just type `C-c C-q` to invoke the command `slime-close-parens-at-point`, which will insert as many closing parentheses as necessary to match all the currently open parentheses.

Now you can get this definition into your Lisp environment in several ways. The easiest is to type `C-c C-c` with the cursor anywhere in or immediately after the **DEFUN** form, which runs the command `slime-compile-defun`, which in turn sends the

definition to Lisp to be evaluated and compiled. To make sure this is working, you can make some change to `hello-world`, recompile it, and then go back to the REPL, using `C-c C-z` or `C-x b`, and call it again. For instance, you could make it a bit more grammatical.

```
(defun hello-world ()
  (format t "Hello, world!"))
```

Next, recompile with `C-c C-c` and then type `C-c C-z` to switch to the REPL to try the new version.

```
CL-USER> (hello-world)
Hello, world!
NIL
```

You'll also probably want to save the file you've been working on; in the `hello.lisp` buffer, type `C-x C-s` to invoke the Emacs command `save-buffer`.

Now to try reloading this function from the source file, you'll need to quit Lisp and restart. To quit you can use a SLIME shortcut: at the REPL, type a comma. At the bottom of the Emacs window, you will be prompted for a command. Type `quit` (or `sayoonara`), and then hit Enter. This will quit Lisp and close all the buffers created by SLIME such as the REPL buffer.[12] Now restart SLIME by typing `M-x slime`.

Just for grins, you can try to invoke `hello-world`.

```
CL-USER> (hello-world)
```

At that point SLIME will pop up a new buffer that starts with something that looks like this:

```
attempt to call `HELLO-WORLD' which is an undefined function.
   [Condition of type UNDEFINED-FUNCTION]

Restarts:
  0: [TRY-AGAIN] Try calling HELLO-WORLD again.
  1: [RETURN-VALUE] Return a value instead of calling HELLO-WORLD.
  2: [USE-VALUE] Try calling a function other than HELLO-WORLD.
  3: [STORE-VALUE] Setf the symbol-function of HELLO-WORLD and call it again.
  4: [ABORT] Abort handling SLIME request.
  5: [ABORT] Abort entirely from this process.

Backtrace:
  0: (SWANK::DEBUG-IN-EMACS #<UNDEFINED-FUNCTION @ #x716b082a>)
  1: ((FLET SWANK:SWANK-DEBUGGER-HOOK SWANK::DEBUG-IT))
  2: (SWANK:SWANK-DEBUGGER-HOOK #<UNDEFINED-FUNCTION @ #x716b082a> #<Function SWANK-DEBUGGER-HOOK>)
  3: (ERROR #<UNDEFINED-FUNCTION @ #x716b082a>)
  4: (EVAL (HELLO-WORLD))
  5: (SWANK::EVAL-REGION "(hello-world)
" T)
```

Blammo! What happened? Well, you tried to invoke a function that doesn't exist. But despite the burst of output, Lisp is actually handling this situation gracefully. Unlike Java or Python, Common Lisp doesn't just bail--throwing an exception and unwinding the stack. And it definitely doesn't dump core just because you tried to invoke a missing function. Instead Lisp drops you into the debugger.

While you're in the debugger you still have full access to Lisp, so you can evaluate expressions to examine the state of our program and maybe even fix things. For now don't worry about that; just type `q` to exit the debugger and get back to the REPL. The debugger buffer will go away, and the REPL will show this:

```
CL-USER> (hello-world)
; Evaluation aborted
CL-USER>
```

There's obviously more that can be done from within the debugger than just abort--we'll see, for instance, in Chapter 19 how the debugger integrates with the error handling system. For now, however, the important thing to know is that you can always get out of it, and back to the REPL, by typing `q`.

Back at the REPL you can try again. Things blew up because Lisp didn't know the definition of `hello-world`. So you need to let Lisp know about the definition we saved in the file `hello.lisp`. You have several ways you could do this. You could switch back to the buffer containing the file (type `C-x b` and then enter `hello.lisp` when prompted) and recompile the definition as you did before with `C-c C-c`. Or you can load the whole file, which would be a more convenient approach if the file contained a bunch of definitions, using the `LOAD` function at the REPL like this:

```
CL-USER> (load "hello.lisp")
; Loading /home/peter/my-lisp-programs/hello.lisp
T
```

The **`T`** means everything loaded correctly.[13] Loading a file with `LOAD` is essentially equivalent to typing each of the expressions in the file at the REPL in the order they appear in the file, so after the call to **`LOAD`**, `hello-world` should be defined:

```
CL-USER> (hello-world)
Hello, world!
NIL
```

Another way to load a file's worth of definitions is to compile the file first with `COMPILE-FILE` and then `LOAD` the resulting compiled file, called a *FASL file*, which is short for *fast-load file*. `COMPILE-FILE` returns the name of the FASL file, so we can compile and load from the REPL like this:

```
CL-USER> (load (compile-file "hello.lisp"))
;;; Compiling file hello.lisp
;;; Writing fasl file hello.fasl
;;; Fasl write complete
; Fast loading /home/peter/my-lisp-programs/hello.fasl
T
```

SLIME also provides support for loading and compiling files without using the REPL. When you're in a source code buffer, you can use `C-c C-l` to load the file with `slime-load-file`. Emacs will prompt for the name of a file to load with the name of the current file already filled in; you can just hit Enter. Or you can type `C-c C-k` to compile and load the file represented by the current buffer. In some Common Lisp implementations, compiling code this way will make it quite a bit faster; in others, it won't, typically because they always compile everything.

This should be enough to give you a flavor of how Lisp programming works. Of course I haven't covered all the tricks and techniques yet, but you've seen the essential elements--interacting with the REPL trying things out, loading and testing new code, tweaking and debugging. Serious Lisp hackers often keep a Lisp image running for days on end, adding, redefining, and testing bits of their program incrementally.

Also, even when the Lisp app is deployed, there's often still a way to get to a REPL. You'll see in Chapter 26 how you can use the REPL and SLIME to interact with the Lisp that's running a Web server at the same time as it's serving up Web pages. It's even possible to use SLIME to connect to a Lisp running on a different machine, allowing you--for instance--to debug a remote server just like a local one.

An even more impressive instance of remote debugging occurred on NASA's 1998 Deep Space 1 mission. A half year after the space craft launched, a bit of Lisp code was going to control the spacecraft for two days while conducting a sequence of experiments. Unfortunately, a subtle race condition in the code had escaped detection during ground testing and was already in space. When the bug manifested in the wild--100 million miles away from Earth--the team was able to

diagnose and fix the running code, allowing the experiments to complete.[14] One of the programmers described it as follows:

> Debugging a program running on a $100M piece of hardware that is 100 million miles away is an interesting experience. Having a read-eval-print loop running on the spacecraft proved invaluable in finding and fixing the problem.

You're not quite ready to send any Lisp code into deep space, but in the next chapter you'll take a crack at writing a program a bit more interesting than "hello, world."

---

[1]Superior Lisp Interaction Mode for Emacs

[2]If you've had a bad experience with Emacs previously, you should treat Lisp in a Box as an IDE that happens to use an Emacs-like editor as its text editor; there will be no need to become an Emacs guru to program Lisp. It is, however, orders of magnitude more enjoyable to program Lisp with an editor that has some basic Lisp awareness. At a minimum, you'll want an editor that can automatically match `()`s for you and knows how to automatically indent Lisp code. Because Emacs is itself largely written in a Lisp dialect, Elisp, it has quite a bit of support for editing Lisp code. Emacs is also deeply embedded into the history of Lisp and the culture of Lisp hackers: the original Emacs and its immediate predecessors, TECMACS and TMACS, were written by Lispers at the Massachusetts Institute of Technology (MIT). The editors on the Lisp Machines were versions of Emacs written entirely in Lisp. The first two Lisp Machine Emacs, following the hacker tradition of recursive acronyms, were EINE and ZWEI, which stood for EINE Is Not Emacs and ZWEI Was EINE Initially. Later ones used a descendant of ZWEI, named, more prosaically, ZMACS.

[3]Practically speaking, there's very little likelihood of the language standard itself being revised--while there are a small handful of warts that folks might like to clean up, the ANSI process isn't amenable to opening an existing standard for minor tweaks, and none of the warts that might be cleaned up actually cause anyone any serious difficulty. The future of Common Lisp standardization is likely to proceed via de facto standards, much like the "standardization" of Perl and Python--as different implementers experiment with application programming interfaces (APIs) and libraries for doing things not specified in the language standard, other implementers may adopt them or people will develop portability libraries to smooth over the differences between implementations for features not specified in the language standard.

[4]Steel Bank Common Lisp

[5]CMU Common Lisp

[6]SBCL forked from CMUCL in order to focus on cleaning up the internals and making it easier to maintain. But the fork has been amiable; bug fixes tend to propagate between the two projects, and there's talk that someday they will merge back together.

[7]The venerable "hello, world" predates even the classic Kernighan and Ritchie C book that played a big role in its popularization. The original "hello, world" seems to have come from Brian Kernighan's "A Tutorial Introduction to the Language B" that was part of the *Bell Laboratories Computing Science Technical Report*

#8: The Programming Language B published in January 1973. (It's available online at `http://cm.bell-labs.com/cm/cs/who/dmr/bintro.html`.)

[8]These are some other expressions that also print the string "hello, world":

[9]Well, as you'll see when I discuss returning multiple values, it's technically possible to write expressions that evaluate to no value, but even such expressions are treated as returning `NIL` when evaluated in a context that expects a value.

[10]I'll discuss in Chapter 4 why the name has been converted to all uppercase.

[11]You could also have entered the definition as two lines at the REPL, as the REPL reads whole expressions, not lines.

[12]SLIME shortcuts aren't part of Common Lisp--they're commands to SLIME.

[13]If for some reason the `LOAD` doesn't go cleanly, you'll get another error and drop back into the debugger. If this happens, the most likely reason is that Lisp can't find the file, probably because its idea of the current working directory isn't the same as where the file is located. In that case, you can quit the debugger by typing `q` and then use the SLIME shortcut `cd` to change Lisp's idea of the current directory--type a comma and then `cd` when prompted for a command and then the name of the directory where `hello.lisp` was saved.

[14]`http://www.flownet.com/gat/jpl-lisp.html`

Copyright Š 2003-2005, Peter Seibel

3. Practical: A Simple Database

Obviously, before you can start building real software in Lisp, you'll have to learn the language. But let's face it--you may be thinking, "'Practical Common Lisp,' isn't that an oxymoron? Why should you be expected to bother learning all the details of a language unless it's actually good for something you care about?" So I'll start by giving you a small example of what you can do with Common Lisp. In this chapter you'll write a simple database for keeping track of CDs. You'll use similar techniques in Chapter 27 when you build a database of MP3s for our streaming MP3 server. In fact, you could think of this as part of the MP3 software project--after all, in order to have a bunch of MP3s to listen to, it might be helpful to be able to keep track of which CDs you have and which ones you need to rip.

In this chapter, I'll cover just enough Lisp as we go along for you to understand how the code works. But I'll gloss over quite a few details. For now you needn't sweat the small stuff--the next several chapters will cover all the Common Lisp constructs used here, and more, in a much more systematic way.

One terminology note: I'll discuss a handful of Lisp operators in this chapter. In Chapter 4, you'll learn that Common Lisp provides three distinct kinds of operators: *functions*, *macros*, and *special operators*. For the purposes of this chapter, you don't really need to know the difference. I will, however, refer to different operators as functions or macros or special operators as appropriate, rather than trying to hide

the details behind the word *operator*. For now you can treat *function*, *macro*, and *special operator* as all more or less equivalent.[1]

Also, keep in mind that I won't bust out all the most sophisticated Common Lisp techniques for your very first post-"hello, world" program. The point of this chapter isn't that this is how you would write a database in Lisp; rather, the point is for you to get an idea of what programming in Lisp is like and to see how even a relatively simple Lisp program can be quite featureful.

CDs and Records

To keep track of CDs that need to be ripped to MP3s and which CDs should be ripped first, each record in the database will contain the title and artist of the CD, a rating of how much the user likes it, and a flag saying whether it has been ripped. So, to start with, you'll need a way to represent a single database record (in other words, one CD). Common Lisp gives you lots of choices of data structures from a simple four-item list to a user-defined class, using the Common Lisp Object System (CLOS).

For now you can stay at the simple end of the spectrum and use a list. You can make a list with the `LIST` function, which, appropriately enough, returns a list of its arguments.

```
CL-USER> (list 1 2 3)
(1 2 3)
```

You could use a four-item list, mapping a given position in the list to a given field in the record. However, another flavor of list--called a *property list*, or *plist* for short--is even more convenient. A plist is a list where every other element, starting with the first, is a *symbol* that describes what the next element in the list is. I won't get into all the details of exactly what a symbol is right now; basically it's a name. For the symbols that name the fields in the CD database, you can use a particular kind of symbol, called a *keyword* symbol. A keyword is any name that starts with a colon (`:`), for instance, `:foo`. Here's an example of a plist using the keyword symbols `:a`, `:b`, and `:c` as property names:

```
CL-USER> (list :a 1 :b 2 :c 3)
(:A 1 :B 2 :C 3)
```

Note that you can create a property list with the same **LIST** function as you use to create other lists; it's the contents that make it a plist.

The thing that makes plists a convenient way to represent the records in a database is the function **GETF**, which takes a plist and a symbol and returns the value in the plist following the symbol, making a plist a sort of poor man's hash table. Lisp has real hash tables too, but plists are sufficient for your needs here and can more easily be saved to a file, which will come in handy later.

```
CL-USER> (getf (list :a 1 :b 2 :c 3) :a)
1
CL-USER> (getf (list :a 1 :b 2 :c 3) :c)
3
```

Given all that, you can easily enough write a function `make-cd` that will take the four fields as arguments and return a plist representing that CD.

```
(defun make-cd (title artist rating ripped)
  (list :title title :artist artist :rating rating :ripped ripped))
```

The word **DEFUN** tells us that this form is defining a new function. The name of the function is `make-cd`. After the name comes the parameter list. This function has four parameters: `title`, `artist`, `rating`, and `ripped`. Everything after the parameter list is the body of the function. In this case the body is just one form, a call to **LIST**. When `make-cd` is called, the arguments passed to the call will be bound to the variables in the parameter list. For instance, to make a record for the CD *Roses* by Kathy Mattea, you might call `make-cd` like this:

```
CL-USER> (make-cd "Roses" "Kathy Mattea" 7 t)
(:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T)
```

Filing CDs

A single record, however, does not a database make. You need some larger construct to hold the records. Again, for simplicity's sake, a list seems like a good choice. Also for simplicity you can use a global variable, `*db*`, which you can define with the **DEFVAR** macro. The asterisks (*) in the name are a Lisp naming convention for global variables.[2]

```
(defvar *db* nil)
```

You can use the **PUSH** macro to add items to `*db*`. But it's probably a good idea to abstract things a tiny bit, so you should define a function `add-record` that adds a record to the database.

```
(defun add-record (cd) (push cd *db*))
```

Now you can use `add-record` and `make-cd` together to add CDs to the database.

```
CL-USER> (add-record (make-cd "Roses" "Kathy Mattea" 7 t))
((:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
CL-USER> (add-record (make-cd "Fly" "Dixie Chicks" 8 t))
((:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
CL-USER> (add-record (make-cd "Home" "Dixie Chicks" 9 t))
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
```

The stuff printed by the REPL after each call to `add-record` is the return value, which is the value returned by the last expression in the function body, the **PUSH**. And **PUSH** returns the new value of the variable it's modifying. So what you're actually seeing is the value of the database after the record has been added.

Looking at the Database Contents

You can also see the current value of `*db*` whenever you want by typing `*db*` at the REPL.

```
CL-USER> *db*
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
```

However, that's not a very satisfying way of looking at the output. You can write a `dump-db` function that dumps out the database in a more human-readable format, like this:

```
TITLE:    Home
ARTIST:   Dixie Chicks
RATING:   9
RIPPED:   T

TITLE:    Fly
ARTIST:   Dixie Chicks
RATING:   8
RIPPED:   T

TITLE:    Roses
ARTIST:   Kathy Mattea
RATING:   7
RIPPED:   T
```

The function looks like this:

```
(defun dump-db ()
  (dolist (cd *db*)
    (format t "~{~a:~10t~a~%~}~%" cd)))
```

This function works by looping over all the elements of `*db*` with the **DOLIST** macro, binding each element to the variable `cd` in turn. For each value of `cd`, you use the **FORMAT** function to print it.

Admittedly, the **FORMAT** call is a little cryptic. However, **FORMAT** isn't particularly more complicated than C or Perl's `printf` function or

Python's string-`%` operator. In Chapter 18 I'll discuss **FORMAT** in greater detail. For now we can take this call bit by bit. As you saw in Chapter 2, **FORMAT** takes at least two arguments, the first being the stream where it sends its output; `t` is shorthand for the stream `*standard-output*`.

The second argument to **FORMAT** is a format string that can contain both literal text and directives telling **FORMAT** things such as how to interpolate the rest of its arguments. Format directives start with `~` (much the way `printf`'s directives start with `%`). **FORMAT** understands dozens of directives, each with their own set of options.[3] However, for now I'll just focus on the ones you need to write `dump-db`.

The `~a` directive is the *aesthetic* directive; it means to consume one argument and output it in a human-readable form. This will render keywords without the leading : and strings without quotation marks. For instance:

```
CL-USER> (format t "~a" "Dixie Chicks")
Dixie Chicks
NIL
```

or:

```
CL-USER> (format t "~a" :title)
TITLE
NIL
```

The `~t` directive is for tabulating. The `~10t` tells **FORMAT** to emit enough spaces to move to the tenth column before processing the next `~a`. A `~t` doesn't consume any arguments.

```
CL-USER> (format t "~a:~10t~a" :artist "Dixie Chicks")
ARTIST:   Dixie Chicks
NIL
```

Now things get slightly more complicated. When **FORMAT** sees `~{` the next argument to be consumed must be a list. **FORMAT** loops over that list, processing the directives between the `~{` and `~}`, consuming as many elements of the list as needed each time through the list. In `dump-db`, the **FORMAT** loop will consume one keyword and one value from the list each time through the loop. The `~%` directive doesn't consume any arguments but tells **FORMAT** to emit a newline. Then after the `~}` ends the loop, the last `~%` tells **FORMAT** to emit one more newline to put a blank line between each CD.

Technically, you could have also used **FORMAT** to loop over the database itself, turning our `dump-db` function into a one-liner.

```
(defun dump-db ()
  (format t "~{~{~a:~10t~a~%~}~%~}" *db*))
```

That's either very cool or very scary depending on your point of view.

Improving the User Interaction

While our `add-record` function works fine for adding records, it's a bit Lispy for the casual user. And if they want to add a bunch of records, it's not very convenient. So you may want to write a function to prompt the user for information about a set of CDs. Right away you know you'll need some way to prompt the user for a piece of information and read it. So let's write that.

```
(defun prompt-read (prompt)
  (format *query-io* "~a: " prompt)
  (force-output *query-io*)
  (read-line *query-io*))
```

You use your old friend **`FORMAT`** to emit a prompt. Note that there's no `~%` in the format string, so the cursor will stay on the same line. The call to **`FORCE-OUTPUT`** is necessary in some implementations to ensure that Lisp doesn't wait for a newline before it prints the prompt.

Then you can read a single line of text with the aptly named **`READ-LINE`** function. The variable `*query-io*` is a global variable (which you can tell because of the `*` naming convention for global variables) that contains the input stream connected to the terminal. The return value of `prompt-read` will be the value of the last form, the call to **`READ-LINE`**, which returns the string it read (without the trailing newline.)

You can combine your existing `make-cd` function with `prompt-read` to build a function that makes a new CD record from data it gets by prompting for each value in turn.

```
(defun prompt-for-cd ()
  (make-cd
```

```
  (prompt-read "Title")
  (prompt-read "Artist")
  (prompt-read "Rating")
  (prompt-read "Ripped [y/n]")))
```

That's almost right. Except `prompt-read` returns a string, which, while fine for the Title and Artist fields, isn't so great for the Rating and Ripped fields, which should be a number and a boolean. Depending on how sophisticated a user interface you want, you can go to arbitrary lengths to validate the data the user enters. For now let's lean toward the quick and dirty: you can wrap the `prompt-read` for the rating in a call to Lisp's **`PARSE-INTEGER`** function, like this:

```
(parse-integer (prompt-read "Rating"))
```

Unfortunately, the default behavior of **`PARSE-INTEGER`** is to signal an error if it can't parse an integer out of the string or if there's any non-numeric junk in the string. However, it takes an optional keyword argument `:junk-allowed`, which tells it to relax a bit.

```
(parse-integer (prompt-read "Rating") :junk-allowed t)
```

But there's still one problem: if it can't find an integer amidst all the junk, **`PARSE-INTEGER`** will return `NIL` rather than a number. In keeping with the quick-and-dirty approach, you may just want to call that 0 and continue. Lisp's **`OR`** macro is just the thing you need here. It's similar to the "short-circuiting" `||` in Perl, Python, Java, and C; it takes a series of expressions, evaluates them one at a time, and returns the first non-nil value (or **`NIL`** if they're all **`NIL`**). So you can use the following:

```
(or (parse-integer (prompt-read "Rating") :junk-allowed t) 0)
```

to get a default value of 0.

Fixing the code to prompt for Ripped is quite a bit simpler. You can just use the Common Lisp function **Y-OR-N-P**.

```
(y-or-n-p "Ripped [y/n]: ")
```

In fact, this will be the most robust part of `prompt-for-cd`, as **Y-OR-N-P** will reprompt the user if they enter something that doesn't start with *y*, *Y*, *n*, or *N*.

Putting those pieces together you get a reasonably robust `prompt-for-cd` function.

```
(defun prompt-for-cd ()
  (make-cd
   (prompt-read "Title")
   (prompt-read "Artist")
   (or (parse-integer (prompt-read "Rating") :junk-allowed t) 0)
   (y-or-n-p "Ripped [y/n]: ")))
```

Finally, you can finish the "add a bunch of CDs" interface by wrapping `prompt-for-cd` in a function that loops until the user is done. You can use the simple form of the **LOOP** macro, which repeatedly executes a body of expressions until it's exited by a call to **RETURN**. For example:

```
(defun add-cds ()
  (loop (add-record (prompt-for-cd))
      (if (not (y-or-n-p "Another? [y/n]: ")) (return))))
```

Now you can use `add-cds` to add some more CDs to the database.

```
CL-USER> (add-cds)
Title: Rockin' the Suburbs
Artist: Ben Folds
Rating: 6
Ripped  [y/n]: y
Another?  [y/n]: y
Title: Give Us a Break
Artist: Limpopo
Rating: 10
Ripped  [y/n]: y
Another?  [y/n]: y
Title: Lyle Lovett
```

Artist: Lyle Lovett
Rating: 9
Ripped [y/n]: y
Another? [y/n]: n
NIL

Saving and Loading the Database

Having a convenient way to add records to the database is nice. But it's not so nice that the user is going to be very happy if they have to reenter all the records every time they quit and restart Lisp. Luckily, with the data structures you're using to represent the data, it's trivially easy to save the data to a file and reload it later. Here's a `save-db` function that takes a filename as an argument and saves the current state of the database:

```
(defun save-db (filename)
  (with-open-file (out filename
                   :direction :output
                   :if-exists :supersede)
    (with-standard-io-syntax
      (print *db* out))))
```

The **`WITH-OPEN-FILE`** macro opens a file, binds the stream to a variable, executes a set of expressions, and then closes the file. It also makes sure the file is closed even if something goes wrong while evaluating the body. The list directly after **`WITH-OPEN-FILE`** isn't a function call but rather part of the syntax defined by **`WITH-OPEN-FILE`**. It contains the name of the variable that will hold the file stream to which you'll write within the body of **`WITH-OPEN-FILE`**, a value that must be a file name, and then some options that control how the file is opened. Here you specify that you're opening the file for writing with `:direction :output` and that you want to overwrite an existing file of the same name if it exists with `:if-exists :supersede`.

Once you have the file open, all you have to do is print the contents of the database with `(print *db* out)`. Unlike **`FORMAT`**, **`PRINT`** prints Lisp objects in a form that can be read back in by the Lisp reader.

The macro **WITH-STANDARD-IO-SYNTAX** ensures that certain variables that affect the behavior of **PRINT** are set to their standard values. You'll use the same macro when you read the data back in to make sure the Lisp reader and printer are operating compatibly.

The argument to `save-db` should be a string containing the name of the file where the user wants to save the database. The exact form of the string will depend on what operating system they're using. For instance, on a Unix box they should be able to call `save-db` like this:

```
CL-USER> (save-db "~/my-cds.db")
((:TITLE "Lyle Lovett" :ARTIST "Lyle Lovett" :RATING 9 :RIPPED T)
 (:TITLE "Give Us a Break" :ARTIST "Limpopo" :RATING 10 :RIPPED T)
 (:TITLE "Rockin' the Suburbs" :ARTIST "Ben Folds" :RATING 6 :RIPPED
  T)
 (:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 9 :RIPPED T))
```

On Windows, the filename might be something like "`c:/my-cds.db`" or "`c:\\my-cds.db`."[4]

You can open this file in any text editor to see what it looks like. You should see something a lot like what the REPL prints if you type `*db*`.

The function to load the database back in is similar.

```
(defun load-db (filename)
  (with-open-file (in filename)
    (with-standard-io-syntax
      (setf *db* (read in)))))
```

This time you don't need to specify `:direction` in the options to **WITH-OPEN-FILE**, since you want the default of `:input`. And instead of printing, you use the function **READ** to read from the stream `in`. This is the same reader used by the REPL and can read any Lisp

expression you could type at the REPL prompt. However, in this case, you're just reading and saving the expression, not evaluating it. Again, the **WITH-STANDARD-IO-SYNTAX** macro ensures that **READ** is using the same basic syntax that `save-db` did when it **PRINT**ed the data.

The **SETF** macro is Common Lisp's main assignment operator. It sets its first argument to the result of evaluating its second argument. So in `load-db` the `*db*` variable will contain the object read from the file, namely, the list of lists written by `save-db`. You do need to be careful about one thing--`load-db` clobbers whatever was in `*db*` before the call. So if you've added records with `add-record` or `add-cds` that haven't been saved with `save-db`, you'll lose them.

Querying the Database

Now that you have a way to save and reload the database to go along with a convenient user interface for adding new records, you soon may have enough records that you won't want to be dumping out the whole database just to look at what's in it. What you need is a way to query the database. You might like, for instance, to be able to write something like this:

```
(select :artist "Dixie Chicks")
```

and get a list of all the records where the artist is the Dixie Chicks. Again, it turns out that the choice of saving the records in a list will pay off.

The function **REMOVE-IF-NOT** takes a predicate and a list and returns a list containing only the elements of the original list that match the predicate. In other words, it has removed all the elements that don't match the predicate. However, **REMOVE-IF-NOT** doesn't really remove anything--it creates a new list, leaving the original list untouched. It's like running grep over a file. The predicate argument can be any function that accepts a single argument and returns a boolean value--**NIL** for false and anything else for true.

For instance, if you wanted to extract all the even elements from a list of numbers, you could use **REMOVE-IF-NOT** as follows:

```
CL-USER> (remove-if-not #'evenp '(1 2 3 4 5 6 7 8 9 10))
(2 4 6 8 10)
```

In this case, the predicate is the function **EVENP**, which returns true if its argument is an even number. The funny notation `#'` is shorthand for "Get me the function with the following name." Without the `#'`, Lisp would treat `evenp` as the name of a variable and look up the value of the variable, not the function.

You can also pass **REMOVE-IF-NOT** an anonymous function. For instance, if **EVENP** didn't exist, you could write the previous expression as the following:

```
CL-USER> (remove-if-not #'(lambda (x) (= 0 (mod x 2))) '(1 2 3 4 5 6 7 8 9 10))
(2 4 6 8 10)
```

In this case, the predicate is this anonymous function:

```
(lambda (x) (= 0 (mod x 2)))
```

which checks that its argument is equal to 0 modulus 2 (in other words, is even). If you wanted to extract only the odd numbers using an anonymous function, you'd write this:

```
CL-USER> (remove-if-not #'(lambda (x) (= 1 (mod x 2))) '(1 2 3 4 5 6 7 8 9 10))
(1 3 5 7 9)
```

Note that `lambda` isn't the name of the function--it's the indicator you're defining an anonymous function.[5] Other than the lack of a name, however, a **LAMBDA** expression looks a lot like a **DEFUN**: the word `lambda` is followed by a parameter list, which is followed by the body of the function.

To select all the Dixie Chicks' albums in the database using **REMOVE-IF-NOT**, you need a function that returns true when the artist field of a record is `"Dixie Chicks"`. Remember that we chose the plist representation for the database records because the function

GETF can extract named fields from a plist. So assuming `cd` is the name of a variable holding a single database record, you can use the expression `(getf cd :artist)` to extract the name of the artist. The function **EQUAL**, when given string arguments, compares them character by character. So `(equal (getf cd :artist) "Dixie Chicks")` will test whether the artist field of a given CD is equal to `"Dixie Chicks"`. All you need to do is wrap that expression in a **LAMBDA** form to make an anonymous function and pass it to **REMOVE-IF-NOT**.

```
CL-USER> (remove-if-not
  #'(lambda (cd) (equal (getf cd :artist) "Dixie Chicks")) *db*)
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T))
```

Now suppose you want to wrap that whole expression in a function that takes the name of the artist as an argument. You can write that like this:

```
(defun select-by-artist (artist)
  (remove-if-not
   #'(lambda (cd) (equal (getf cd :artist) artist))
   *db*))
```

Note how the anonymous function, which contains code that won't run until it's invoked in **REMOVE-IF-NOT**, can nonetheless refer to the variable `artist`. In this case the anonymous function doesn't just save you from having to write a regular function--it lets you write a function that derives part of its meaning--the value of `artist`--from the context in which it's embedded.

So that's `select-by-artist`. However, selecting by artist is only one of the kinds of queries you might like to support. You *could* write several more functions, such as `select-by-title`, `select-by-rating`,

`select-by-title-and-artist`, and so on. But they'd all be about the same except for the contents of the anonymous function. You can instead make a more general `select` function that takes a function as an argument.

```
(defun select (selector-fn)
  (remove-if-not selector-fn *db*))
```

So what happened to the `#'`? Well, in this case you don't want **`REMOVE-IF-NOT`** to use the function named `selector-fn`. You want it to use the anonymous function that was passed as an argument to `select` in the *variable* `selector-fn`. Though, the `#'` comes back in the *call* to `select`.

```
CL-USER> (select #'(lambda (cd) (equal (getf cd :artist) "Dixie Chicks")))
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T))
```

But that's really quite gross-looking. Luckily, you can wrap up the creation of the anonymous function.

```
(defun artist-selector (artist)
  #'(lambda (cd) (equal (getf cd :artist) artist)))
```

This is a function that returns a function and one that references a variable that--it seems--won't exist after `artist-selector` returns.[6] It may seem odd now, but it actually works just the way you'd want--if you call `artist-selector` with an argument of `"Dixie Chicks"`, you get an anonymous function that matches CDs whose `:artist` field is `"Dixie Chicks"`, and if you call it with `"Lyle Lovett"`, you get a different function that will match against an `:artist` field of `"Lyle Lovett"`. So now you can rewrite the call to `select` like this:

```
CL-USER> (select (artist-selector "Dixie Chicks"))
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T))
```

Now you just need some more functions to generate selectors. But just as you don't want to have to write `select-by-title`, `select-by-rating`, and so on, because they would all be quite similar, you're not going to want to write a bunch of nearly identical selector-function generators, one for each field. Why not write one general-purpose selector-function generator, a function that, depending on what arguments you pass it, will generate a selector function for different fields or maybe even a combination of fields? You can write such a function, but first you need a crash course in a feature called *keyword parameters*.

In the functions you've written so far, you've specified a simple list of parameters, which are bound to the corresponding arguments in the call to the function. For instance, the following function:

```
(defun foo (a b c) (list a b c))
```

has three parameters, `a`, `b`, and `c`, and must be called with three arguments. But sometimes you may want to write a function that can be called with varying numbers of arguments. Keyword parameters are one way to achieve this. A version of `foo` that uses keyword parameters might look like this:

```
(defun foo (&key a b c) (list a b c))
```

The only difference is the `&key` at the beginning of the argument list. However, the calls to this new `foo` will look quite different. These are all legal calls with the result to the right of the ==>:

```
(foo :a 1 :b 2 :c 3)  ==> (1 2 3)
(foo :c 3 :b 2 :a 1)  ==> (1 2 3)
(foo :a 1 :c 3)       ==> (1 NIL 3)
(foo)                 ==> (NIL NIL NIL)
```

As these examples show, the value of the variables `a`, `b`, and `c` are bound to the values that follow the corresponding keyword. And if a particular keyword isn't present in the call, the corresponding variable is set to **NIL**. I'm glossing over a bunch of details of how keyword parameters are specified and how they relate to other kinds of parameters, but you need to know one more detail.

Normally if a function is called with no argument for a particular keyword parameter, the parameter will have the value **NIL**. However, sometimes you'll want to be able to distinguish between a **NIL** that was explicitly passed as the argument to a keyword parameter and the default value **NIL**. To allow this, when you specify a keyword parameter you can replace the simple name with a list consisting of the name of the parameter, a default value, and another parameter name, called a *supplied-p* parameter. The supplied-p parameter will be set to true or false depending on whether an argument was actually passed for that keyword parameter in a particular call to the function. Here's a version of `foo` that uses this feature:

```
(defun foo (&key a (b 20) (c 30 c-p)) (list a b c c-p))
```

Now the same calls from earlier yield these results:

```
(foo :a 1 :b 2 :c 3)  ==> (1 2 3 T)
(foo :c 3 :b 2 :a 1)  ==> (1 2 3 T)
(foo :a 1 :c 3)       ==> (1 20 3 T)
(foo)                 ==> (NIL 20 30 NIL)
```

The general selector-function generator, which you can call `where` for reasons that will soon become apparent if you're familiar with SQL databases, is a function that takes four keyword parameters corresponding to the fields in our CD records and generates a

selector function that selects any CDs that match all the values given to `where`. For instance, it will let you say things like this:

```
(select (where :artist "Dixie Chicks"))
```

or this:

```
(select (where :rating 10 :ripped nil))
```

The function looks like this:

```
(defun where (&key title artist rating (ripped nil ripped-p))
  #'(lambda (cd)
      (and
       (if title    (equal (getf cd :title)  title)  t)
       (if artist   (equal (getf cd :artist) artist) t)
       (if rating   (equal (getf cd :rating) rating) t)
       (if ripped-p (equal (getf cd :ripped) ripped) t))))
```

This function returns an anonymous function that returns the logical `AND` of one clause per field in our CD records. Each clause checks if the appropriate argument was passed in and then either compares it to the value in the corresponding field in the CD record or returns `t`, Lisp's version of truth, if the parameter wasn't passed in. Thus, the selector function will return `t` only for CDs that match all the arguments passed to `where`.[7] Note that you need to use a three-item list to specify the keyword parameter `ripped` because you need to know whether the caller actually passed `:ripped nil`, meaning, "Select CDs whose ripped field is nil," or whether they left out `:ripped` altogether, meaning "I don't care what the value of the ripped field is."

Updating Existing Records--Another Use for WHERE

Now that you've got nice generalized `select` and `where` functions, you're in a good position to write the next feature that every database needs--a way to update particular records. In SQL the `update` command is used to update a set of records matching a particular `where` clause. That seems like a good model, especially since you've already got a `where`-clause generator. In fact, the `update` function is mostly just the application of a few ideas you've already seen: using a passed-in selector function to choose the records to update and using keyword arguments to specify the values to change. The main new bit is the use of a function **MAPCAR** that maps over a list, `*db*` in this case, and returns a new list containing the results of calling a function on each item in the original list.

```
(defun update (selector-fn &key title artist rating (ripped nil ripped-p))
  (setf *db*
        (mapcar
         #'(lambda (row)
             (when (funcall selector-fn row)
               (if title    (setf (getf row :title) title))
               (if artist   (setf (getf row :artist) artist))
               (if rating   (setf (getf row :rating) rating))
               (if ripped-p (setf (getf row :ripped) ripped)))
             row) *db*)))
```

One other new bit here is the use of **SETF** on a complex form such as `(getf row :title)`. I'll discuss **SETF** in greater detail in Chapter 6, but for now you just need to know that it's a general assignment operator that can be used to assign lots of "places" other than just variables. (It's a coincidence that **SETF** and **GETF** have such similar names--they don't have any special relationship.) For now it's enough to know that after `(setf (getf row :title) title)`, the plist referenced by row

will have the value of the variable `title` following the property name `:title`. With this `update` function if you decide that you *really* dig the Dixie Chicks and that all their albums should go to 11, you can evaluate the following form:

```
CL-USER> (update (where :artist "Dixie Chicks") :rating 11)
NIL
```

And it is so.

```
CL-USER> (select (where :artist "Dixie Chicks"))
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 11 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 11 :RIPPED T))
```

You can even more easily add a function to delete rows from the database.

```
(defun delete-rows (selector-fn)
  (setf *db* (remove-if selector-fn *db*)))
```

The function **REMOVE-IF** is the complement of **REMOVE-IF-NOT**; it returns a list with all the elements that do match the predicate removed. Like **REMOVE-IF-NOT**, it doesn't actually affect the list it's passed but by saving the result back into `*db*`, `delete-rows`[8] actually changes the contents of the database.[9]

Removing Duplication and Winning Big

So far *all* the database code supporting insert, select, update, and delete, not to mention a command-line user interface for adding new records and dumping out the contents, is just a little more than 50 lines. Total.[10]

Yet there's still some annoying code duplication. And it turns out you can remove the duplication and make the code more flexible at the same time. The duplication I'm thinking of is in the where function. The body of the `where` function is a bunch of clauses like this, one per field:

```
(if title (equal (getf cd :title) title) t)
```

Right now it's not so bad, but like all code duplication it has the same cost: if you want to change how it works, you have to change multiple copies. And if you change the fields in a CD, you'll have to add or remove clauses to `where`. And `update` suffers from the same kind of duplication. It's doubly annoying since the whole point of the `where` function is to dynamically generate a bit of code that checks the values you care about; why should it have to do work at runtime checking whether `title` was even passed in?

Imagine that you were trying to optimize this code and discovered that it was spending too much time checking whether `title` and the rest of the keyword parameters to `where` were even set?[11] If you really wanted to remove all those runtime checks, you could go through a program and find all the places you call `where` and look at exactly what arguments you're passing. Then you could replace

each call to `where` with an anonymous function that does only the computation necessary. For instance, if you found this snippet of code:

```
(select (where :title "Give Us a Break" :ripped t))
```

you could change it to this:

```
(select
 #'(lambda (cd)
     (and (equal (getf cd :title) "Give Us a Break")
          (equal (getf cd :ripped) t))))
```

Note that the anonymous function is different from the one that `where` would have returned; you're not trying to save the call to `where` but rather to provide a more efficient selector function. This anonymous function has clauses only for the fields that you actually care about at this call site, so it doesn't do any extra work the way a function returned by `where` might.

You can probably imagine going through all your source code and fixing up all the calls to `where` in this way. But you can probably also imagine that it would be a huge pain. If there were enough of them, and it was important enough, it might even be worthwhile to write some kind of preprocessor that converts `where` calls to the code you'd write by hand.

The Lisp feature that makes this trivially easy is its macro system. I can't emphasize enough that the Common Lisp macro shares essentially nothing but the name with the text-based macros found in C and C++. Where the C pre-processor operates by textual substitution and understands almost nothing of the structure of C and C++, a Lisp macro is essentially a code generator that gets run

for you automatically by the compiler.[12] When a Lisp expression contains a call to a macro, instead of evaluating the arguments and passing them to the function, the Lisp compiler passes the arguments, unevaluated, to the macro code, which returns a new Lisp expression that is then evaluated in place of the original macro call.

I'll start with a simple, and silly, example and then show how you can replace the `where` function with a `where` macro. Before I can write this example macro, I need to quickly introduce one new function: `REVERSE` takes a list as an argument and returns a new list that is its reverse. So `(reverse '(1 2 3))` evaluates to `(3 2 1)`. Now let's create a macro.

```
(defmacro backwards (expr) (reverse expr))
```

The main syntactic difference between a function and a macro is that you define a macro with **DEFMACRO** instead of **DEFUN**. After that a macro definition consists of a name, just like a function, a parameter list, and a body of expressions, both also like a function. However, a macro has a totally different effect. You can use this macro as follows:

```
CL-USER> (backwards ("hello, world" t format))
hello, world
NIL
```

How did that work? When the REPL started to evaluate the `backwards` expression, it recognized that `backwards` is the name of a macro. So it left the expression `("hello, world" t format)` unevaluated, which is good because it isn't a legal Lisp form. It then passed that list to the `backwards` code. The code in `backwards` passed

the list to **REVERSE**, which returned the list `(format t "hello, world")`. `backwards` then passed that value back out to the REPL, which then evaluated it in place of the original expression.

The `backwards` macro thus defines a new language that's a lot like Lisp--just backward--that you can drop into anytime simply by wrapping a backward Lisp expression in a call to the `backwards` macro. And, in a compiled Lisp program, that new language is just as efficient as normal Lisp because all the macro code--the code that generates the new expression--runs at compile time. In other words, the compiler will generate exactly the same code whether you write `(backwards ("hello, world" t format))` or `(format t "hello, world")`.

So how does that help with the code duplication in `where`? Well, you can write a macro that generates exactly the code you need for each particular call to `where`. Again, the best approach is to build our code bottom up. In the hand-optimized selector function, you had an expression of the following form for each actual field referred to in the original call to `where`:

```
(equal (getf cd field) value)
```

So let's write a function that, given the name of a field and a value, returns such an expression. Since an expression is just a list, you might think you could write something like this:

```
(defun make-comparison-expr (field value)    ; wrong
  (list equal (list getf cd field) value))
```

However, there's one trick here: as you know, when Lisp sees a simple name such as `field` or `value` other than as the first element

of a list, it assumes it's the name of a variable and looks up its value. That's fine for `field` and `value`; it's exactly what you want. But it will treat `equal`, `getf`, and `cd` the same way, which *isn't* what you want. However, you also know how to stop Lisp from evaluating a form: stick a single forward quote (') in front of it. So if you write `make-comparison-expr` like this, it will do what you want:

```
(defun make-comparison-expr (field value)
  (list 'equal (list 'getf 'cd field) value))
```

You can test it out in the REPL.

```
CL-USER> (make-comparison-expr :rating 10)
(EQUAL (GETF CD :RATING) 10)
CL-USER> (make-comparison-expr :title "Give Us a Break")
(EQUAL (GETF CD :TITLE) "Give Us a Break")
```

It turns out that there's an even better way to do it. What you'd really like is a way to write an expression that's mostly not evaluated and then have some way to pick out a few expressions that you *do* want evaluated. And, of course, there's just such a mechanism. A back quote (`) before an expression stops evaluation just like a forward quote.

```
CL-USER> `(1 2 3)
(1 2 3)
CL-USER> '(1 2 3)
(1 2 3)
```

However, in a back-quoted expression, any subexpression that's preceded by a comma is evaluated. Notice the effect of the comma in the second expression:

```
`(1 2 (+ 1 2))        ==> (1 2 (+ 1 2))
`(1 2 ,(+ 1 2))       ==> (1 2 3)
```

Using a back quote, you can write `make-comparison-expr` like this:

```
(defun make-comparison-expr (field value)
  `(equal (getf cd ,field) ,value))
```

Now if you look back to the hand-optimized selector function, you can see that the body of the function consisted of one comparison expression per field/value pair, all wrapped in an **AND** expression. Assume for the moment that you'll arrange for the arguments to the `where` macro to be passed as a single list. You'll need a function that can take the elements of such a list pairwise and collect the results of calling `make-comparison-expr` on each pair. To implement that function, you can dip into the bag of advanced Lisp tricks and pull out the mighty and powerful `LOOP` macro.

```
(defun make-comparisons-list (fields)
  (loop while fields
     collecting (make-comparison-expr (pop fields) (pop fields))))
```

A full discussion of `LOOP` will have to wait until Chapter 22; for now just note that this `LOOP` expression does exactly what you need: it loops while there are elements left in the `fields` list, popping off two at a time, passing them to `make-comparison-expr`, and collecting the results to be returned at the end of the loop. The **POP** macro performs the inverse operation of the `PUSH` macro you used to add records to `*db*`.

Now you just need to wrap up the list returned by `make-comparison-list` in an `AND` and an anonymous function, which you can do in the `where` macro itself. Using a back quote to make a template that you fill in by interpolating the value of `make-comparisons-list`, it's trivial.

```
(defmacro where (&rest clauses)
  `#'(lambda (cd) (and ,@(make-comparisons-list clauses))))
```

This macro uses a variant of `,` (namely, the `,@`) before the call to `make-comparisons-list`. The `,@` "splices" the value of the following expression--which must evaluate to a list--into the enclosing list. You can see the difference between `,` and `,@` in the following two expressions:

```
`(and ,(list 1 2 3))   ==> (AND (1 2 3))
`(and ,@(list 1 2 3))  ==> (AND 1 2 3)
```

You can also use `,@` to splice into the middle of a list.

```
`(and ,@(list 1 2 3) 4) ==> (AND 1 2 3 4)
```

The other important feature of the `where` macro is the use of `&rest` in the argument list. Like `&key`, `&rest` modifies the way arguments are parsed. With a `&rest` in its parameter list, a function or macro can take an arbitrary number of arguments, which are collected into a single list that becomes the value of the variable whose name follows the `&rest`. So if you call `where` like this:

```
(where :title "Give Us a Break" :ripped t)
```

the variable `clauses` will contain the list.

```
(:title "Give Us a Break" :ripped t)
```

This list is passed to `make-comparisons-list`, which returns a list of comparison expressions. You can see exactly what code a call to `where` will generate using the function **`MACROEXPAND-1`**. If you pass **`MACROEXPAND-1`**, a form representing a macro call, it will call the macro code with appropriate arguments and return the expansion. So you can check out the previous `where` call like this:

```
CL-USER> (macroexpand-1 '(where :title "Give Us a Break" :ripped t))
#'(LAMBDA (CD)
    (AND (EQUAL (GETF CD :TITLE) "Give Us a Break")
```

```
          (EQUAL (GETF CD :RIPPED) T)))
T
```

Looks good. Let's try it for real.

```
CL-USER> (select (where :title "Give Us a Break" :ripped t))
((:TITLE "Give Us a Break" :ARTIST "Limpopo" :RATING 10 :RIPPED T))
```

It works. And the `where` macro with its two helper functions is actually one line shorter than the old `where` function. And it's more general in that it's no longer tied to the specific fields in our CD records.

Wrapping Up

Now, an interesting thing has happened. You removed duplication and made the code more efficient *and* more general at the same time. That's often the way it goes with a well-chosen macro. This makes sense because a macro is just another mechanism for creating abstractions--abstraction at the syntactic level, and abstractions are by definition more concise ways of expressing underlying generalities. Now the only code in the mini-database that's specific to CDs and the fields in them is in the `make-cd`, `prompt-for-cd`, and `add-cd` functions. In fact, our new `where` macro would work with any plist-based database.

However, this is still far from being a complete database. You can probably think of plenty of features to add, such as supporting multiple tables or more elaborate queries. In Chapter 27 we'll build an MP3 database that incorporates some of those features.

The point of this chapter was to give you a quick introduction to just a handful of Lisp's features and show how they're used to write code that's a bit more interesting than "hello, world." In the next chapter we'll begin a more systematic overview of Lisp.

---

[1]Before I proceed, however, it's crucially important that you forget anything you may know about #define-style "macros" as implemented in the C pre-processor. Lisp macros are a totally different beast.

[2]Using a global variable also has some drawbacks--for instance, you can have only one database at a time. In Chapter 27, with more of the language under your belt, you'll be

ready to build a more flexible database. You'll also see, in Chapter 6, how even using a global variable is more flexible in Common Lisp than it may be in other languages.

[3]One of the coolest **FORMAT** directives is the `~R` directive. Ever want to know how to say a really big number in English words? Lisp knows. Evaluate this:

```
(format nil "~r" 160693804425899027554196209 2)
```

and you should get back (wrapped for legibility):

> "one octillion six hundred six septillion nine hundred thirty-eight sextillion forty-four quintillion two hundred fifty-eight quadrillion nine hundred ninety trillion two hundred seventy-five billion five hundred forty-one million nine hundred sixty-two thousand ninety-two"

[4]Windows actually understands forward slashes in filenames even though it normally uses a backslash as the directory separator. This is convenient since otherwise you have to write double backslashes because backslash is the escape character in Lisp strings.

[5]The word lambda is used in Lisp because of an early connection to the lambda calculus, a mathematical formalism invented for studying mathematical functions.

[6]The technical term for a function that references a variable in its enclosing scope is a closure because the function "closes over" the variable. I'll discuss closures in more detail in Chapter 6.

[7]Note that in Lisp, an IF form, like everything else, is an expression that returns a value. It's actually more like the ternary operator (`?:`) in Perl, Java, and C in that this is legal in those languages:

```
some_var = some_boolean ? value1 : value2;
```

while this isn't:

```
some_var = if (some_boolean) value1; else value2;
```

because in those languages, `if` is a statement, not an expression.

[8]You need to use the name `delete-rows` rather than the more obvious `delete` because there's already a function in Common Lisp called **DELETE**. The Lisp package system gives you a way to deal with such naming conflicts, so you could have a function named delete if you wanted. But I'm not ready to explain packages just yet.

[9]If you're worried that this code creates a memory leak, rest assured: Lisp was the language that invented garbage collection (and heap allocation for that matter). The memory used by the old value of `*db*` will be automatically reclaimed, assuming no one else is holding on to a reference to it, which none of this code is.

[10]A friend of mine was once interviewing an engineer for a programming job and asked him a typical interview question: how do you know when a function or method is too big? Well, said the candidate, I don't like any method to be bigger than my head. You mean you can't keep all the details in your head? No, I mean I put my head up against my monitor, and the code shouldn't be bigger than my head.

[11]It's unlikely that the cost of checking whether keyword parameters had been passed would be a detectible drag on performance since checking whether a variable is `NIL` is going to be pretty cheap. On the other hand, these functions returned by `where` are going to be right in the middle of the inner loop of any `select`, `update`, or `delete-rows` call, as they have to be called once per entry in the database. Anyway, for illustrative purposes, this will have to do.

[12]Macros are also run by the interpreter--however, it's easier to understand the point of macros when you think about compiled code. As with everything else in this chapter, I'll cover this in greater detail in future chapters.

Copyright © 2003-2005, Peter Seibel

4. Syntax and Semantics

After that whirlwind tour, we'll settle down for a few chapters to take a more systematic look at the features you've used so far. I'll start with an overview of the basic elements of Lisp's syntax and semantics, which means, of course, that I must first address that burning question. . .

What's with All the Parentheses?

Lisp's syntax is quite a bit different from the syntax of languages descended from Algol. The two most immediately obvious characteristics are the extensive use of parentheses and prefix notation. For whatever reason, a lot of folks are put off by this syntax. Lisp's detractors tend to describe the syntax as "weird" and "annoying." Lisp, they say, must stand for Lots of Irritating Superfluous Parentheses. Lisp folks, on the other hand, tend to consider Lisp's syntax one of its great virtues. How is it that what's so off-putting to one group is a source of delight to another?

I can't really make the complete case for Lisp's syntax until I've explained Lisp's macros a bit more thoroughly, but I can start with an historical tidbit that suggests it may be worth keeping an open mind: when John McCarthy first invented Lisp, he intended to implement a more Algol-like syntax, which he called *M-expressions*. However, he never got around to it. He explained why not in his article "History of Lisp."[1]

> The project of defining M-expressions precisely and compiling them or at least translating them into S-expressions was neither finalized nor explicitly abandoned. It just receded into the indefinite future, and a new generation of programmers appeared who preferred [S-expressions] to any FORTRAN-like or ALGOL-like notation that could be devised.

In other words, the people who have actually used Lisp over the past 45 years have *liked* the syntax and have found that it makes the language more powerful. In the next few chapters, you'll begin to see why.

Breaking Open the Black Box

Before we look at the specifics of Lisp's syntax and semantics, it's worth taking a moment to look at how they're defined and how this differs from many other languages.

In most programming languages, the language processor--whether an interpreter or a compiler--operates as a black box: you shove a sequence of characters representing the text of a program into the black box, and it--depending on whether it's an interpreter or a compiler--either executes the behaviors indicated or produces a compiled version of the program that will execute the behaviors when it's run.

Inside the black box, of course, language processors are usually divided into subsystems that are each responsible for one part of the task of translating a program text into behavior or object code. A typical division is to split the processor into three phases, each of which feeds into the next: a lexical analyzer breaks up the stream of characters into tokens and feeds them to a parser that builds a tree representing the expressions in the program, according to the language's grammar. This tree--called an *abstract syntax tree*--is then fed to an evaluator that either interprets it directly or compiles it into some other language such as machine code. Because the language processor is a black box, the data structures used by the processor, such as the tokens and abstract syntax trees, are of interest only to the language implementer.

In Common Lisp things are sliced up a bit differently, with consequences for both the implementer and for how the language is defined. Instead of a single black box that goes from text to program behavior in one step, Common Lisp defines *two* black boxes, one that translates text into Lisp objects and another that implements the semantics of the language in terms of those objects. The first box is called the *reader*, and the second is called the *evaluator*.[2]

Each black box defines one level of syntax. The reader defines how strings of characters can be translated into Lisp objects called *s-expressions*.[3] Since the s-expression syntax includes syntax for lists of arbitrary objects, including other lists, s-expressions can represent arbitrary tree expressions, much like the abstract syntax tree generated by the parsers for non-Lisp languages.

The evaluator then defines a syntax of Lisp *forms* that can be built out of s-expressions. Not all s-expressions are legal Lisp forms any more than all sequences of characters are legal s-expressions. For instance, both `(foo 1 2)` and `("foo" 1 2)` are s-expressions, but only the former can be a Lisp form since a list that starts with a string has no meaning as a Lisp form.

This split of the black box has a couple of consequences. One is that you can use s-expressions, as you saw in Chapter 3, as an externalizable data format for data other than source code, using **READ** to read it and **PRINT** to print it.[4] The other consequence is that since the semantics of the language are defined in terms of trees of objects rather than strings of characters, it's easier to generate code within the language than it would be if you had to generate code as

text. Generating code completely from scratch is only marginally easier--building up lists vs. building up strings is about the same amount of work. The real win, however, is that you can generate code by manipulating existing data. This is the basis for Lisp's macros, which I'll discuss in much more detail in future chapters. For now I'll focus on the two levels of syntax defined by Common Lisp: the syntax of s-expressions understood by the reader and the syntax of Lisp forms understood by the evaluator.

S-expressions

The basic elements of s-expressions are *lists* and *atoms*. Lists are delimited by parentheses and can contain any number of whitespace-separated elements. Atoms are everything else.[5] The elements of lists are themselves s-expressions (in other words, atoms or nested lists). Comments--which aren't, technically speaking, s-expressions--start with a semicolon, extend to the end of a line, and are treated essentially like whitespace.

And that's pretty much it. Since lists are syntactically so trivial, the only remaining syntactic rules you need to know are those governing the form of different kinds of atoms. In this section I'll describe the rules for the most commonly used kinds of atoms: numbers, strings, and names. After that, I'll cover how s-expressions composed of these elements can be evaluated as Lisp forms.

Numbers are fairly straightforward: any sequence of digits--possibly prefaced with a sign (`+` or `-`), containing a decimal point (`.`) or a solidus (`/`), or ending with an exponent marker--is read as a number. For example:

```
123       ; the integer one hundred twenty-three
3/7       ; the ratio three-sevenths
1.0       ; the floating-point number one in default precision
1.0e0     ; another way to write the same floating-point number
1.0d0     ; the floating-point number one in "double" precision
1.0e-4    ; the floating-point equivalent to one-ten-thousandth
+42       ; the integer forty-two
-42       ; the integer negative forty-two
-1/4      ; the ratio negative one-quarter
-2/8      ; another way to write negative one-quarter
246/2     ; another way to write the integer one hundred twenty-three
```

These different forms represent different kinds of numbers: integers, ratios, and floating point. Lisp also supports complex numbers, which have their own notation and which I'll discuss in Chapter 10.

As some of these examples suggest, you can notate the same number in many ways. But regardless of how you write them, all rationals--integers and ratios--are represented internally in "simplified" form. In other words, the objects that represent -2/8 or 246/2 aren't distinct from the objects that represent -1/4 and 123. Similarly, `1.0` and `1.0e0` are just different ways of writing the same number. On the other hand, `1.0`, `1.0d0`, and `1` can all denote different objects because the different floating-point representations and integers are different types. We'll save the details about the characteristics of different kinds of numbers for Chapter 10.

Strings literals, as you saw in the previous chapter, are enclosed in double quotes. Within a string a backslash (`\`) escapes the next character, causing it to be included in the string regardless of what it is. The only two characters that *must* be escaped within a string are double quotes and the backslash itself. All other characters can be included in a string literal without escaping, regardless of their meaning outside a string. Some example string literals are as follows:

```
"foo"     ; the string containing the characters f, o, and o.
"fo\o"    ; the same string
"fo\\o"   ; the string containing the characters f, o, \, and o.
"fo\"o"   ; the string containing the characters f, o, ", and o.
```

Names used in Lisp programs, such as **`FORMAT`** and `hello-world`, and `*db*` are represented by objects called *symbols*. The reader knows nothing about how a given name is going to be used--whether it's the

name of a variable, a function, or something else. It just reads a sequence of characters and builds an object to represent the name.[6] Almost any character can appear in a name. Whitespace characters can't, though, because the elements of lists are separated by whitespace. Digits can appear in names as long as the name as a whole can't be interpreted as a number. Similarly, names can contain periods, but the reader can't read a name that consists only of periods. Ten characters that serve other syntactic purposes can't appear in names: open and close parentheses, double and single quotes, backtick, comma, colon, semicolon, backslash, and vertical bar. And even those characters *can*, if you're willing to escape them by preceding the character to be escaped with a backslash or by surrounding the part of the name containing characters that need escaping with vertical bars.

Two important characteristics of the way the reader translates names to symbol objects have to do with how it treats the case of letters in names and how it ensures that the same name is always read as the same symbol. While reading names, the reader converts all unescaped characters in a name to their uppercase equivalents. Thus, the reader will read `foo`, `Foo`, and `FOO` as the same symbol: `FOO`. However, `\f\o\o` and `|foo|` will both be read as `foo`, which is a different object than the symbol `FOO`. This is why when you define a function at the REPL and it prints the name of the function, it's been converted to uppercase. Standard style, these days, is to write code in all lowercase and let the reader change names to uppercase.[7]

To ensure that the same textual name is always read as the same symbol, the reader *interns* symbols--after it has read the name and

converted it to all uppercase, the reader looks in a table called a *package* for an existing symbol with the same name. If it can't find one, it creates a new symbol and adds it to the table. Otherwise, it returns the symbol already in the table. Thus, anywhere the same name appears in any s-expression, the same object will be used to represent it.[8]

Because names can contain many more characters in Lisp than they can in Algol-derived languages, certain naming conventions are distinct to Lisp, such as the use of hyphenated names like `hello-world`. Another important convention is that global variables are given names that start and end with `*`. Similarly, constants are given names starting and ending in `+`. And some programmers will name particularly low-level functions with names that start with `%` or even `%%`. The names defined in the language standard use only the alphabetic characters (A-Z) plus `*`, `+`, `-`, `/`, `1`, `2`, `<`, `=`, `>`, and `&`.

The syntax for lists, numbers, strings, and symbols can describe a good percentage of Lisp programs. Other rules describe notations for literal vectors, individual characters, and arrays, which I'll cover when I talk about the associated data types in Chapters 10 and 11. For now the key thing to understand is how you can combine numbers, strings, and symbols with parentheses-delimited lists to build s-expressions representing arbitrary trees of objects. Some simple examples look like this:

```
x             ; the symbol X
()            ; the empty list
(1 2 3)       ; a list of three numbers
("foo" "bar") ; a list of two strings
(x y z)       ; a list of three symbols
(x 1 "foo")   ; a list of a symbol, a number, and a string
(+ (* 2 3) 4) ; a list of a symbol, a list, and a number.
```

An only slightly more complex example is the following four-item list that contains two symbols, the empty list, and another list, itself containing two symbols and a string:

```
(defun hello-world ()
  (format t "hello, world"))
```

S-expressions As Lisp Forms

After the reader has translated a bunch of text into s-expressions, the s-expressions can then be evaluated as Lisp code. Or some of them can--not every s-expressions that the reader can read can necessarily be evaluated as Lisp code. Common Lisp's evaluation rule defines a second level of syntax that determines which s-expressions can be treated as Lisp forms.[9] The syntactic rules at this level are quite simple. Any atom--any nonlist or the empty list--is a legal Lisp form as is any list that has a symbol as its first element.[10]

Of course, the interesting thing about Lisp forms isn't their syntax but how they're evaluated. For purposes of discussion, you can think of the evaluator as a function that takes as an argument a syntactically well-formed Lisp form and returns a value, which we can call the *value* of the form. Of course, when the evaluator is a compiler, this is a bit of a simplification--in that case, the evaluator is given an expression and generates code that will compute the appropriate value when it's run. But this simplification lets me describe the semantics of Common Lisp in terms of how the different kinds of Lisp forms are evaluated by this notional function.

The simplest Lisp forms, atoms, can be divided into two categories: symbols and everything else. A symbol, evaluated as a form, is considered the name of a variable and evaluates to the current value of the variable.[11] I'll discuss in Chapter 6 how variables get their values in the first place. You should also note that certain "variables" are that old oxymoron of programming: "constant variables." For instance, the symbol `PI` names a constant variable whose value is

the best possible floating-point approximation to the mathematical constant *pi*.

All other atoms--numbers and strings are the kinds you've seen so far--are *self-evaluating* objects. This means when such an expression is passed to the notional evaluation function, it's simply returned. You saw examples of self-evaluating objects in Chapter 2 when you typed `10` and `"hello, world"` at the REPL.

It's also possible for symbols to be self-evaluating in the sense that the variables they name can be assigned the value of the symbol itself. Two important constants that are defined this way are `T` and `NIL`, the canonical true and false values. I'll discuss their role as booleans in the section "Truth, Falsehood, and Equality."

Another class of self-evaluating symbols are the *keyword* symbols--symbols whose names start with `:`. When the reader interns such a name, it automatically defines a constant variable with the name and with the symbol as the value.

Things get more interesting when we consider how lists are evaluated. All legal list forms start with a symbol, but three kinds of list forms are evaluated in three quite different ways. To determine what kind of form a given list is, the evaluator must determine whether the symbol that starts the list is the name of a function, a macro, or a special operator. If the symbol hasn't been defined yet--as may be the case if you're compiling code that contains references to functions that will be defined later--it's assumed to be a function name.[12] I'll refer to the three kinds of forms as *function call forms*, *macro forms*, and *special forms*.

Function Calls

The evaluation rule for function call forms is simple: evaluate the remaining elements of the list as Lisp forms and pass the resulting values to the named function. This rule obviously places some additional syntactic constraints on a function call form: all the elements of the list after the first must themselves be well-formed Lisp forms. In other words, the basic syntax of a function call form is as follows, where each of the arguments is itself a Lisp form:

```
(function-name argument*)
```

Thus, the following expression is evaluated by first evaluating `1`, then evaluating `2`, and then passing the resulting values to the `+` function, which returns 3:

```
(+ 1 2)
```

A more complex expression such as the following is evaluated in similar fashion except that evaluating the arguments `(+ 1 2)` and `(- 3 4)` entails first evaluating their arguments and applying the appropriate functions to them:

```
(* (+ 1 2) (- 3 4))
```

Eventually, the values 3 and -1 are passed to the `*` function, which returns -3.

As these examples show, functions are used for many of the things that require special syntax in other languages. This helps keep Lisp's syntax regular.

Special Operators

That said, not all operations can be defined as functions. Because all the arguments to a function are evaluated before the function is called, there's no way to write a function that behaves like the **IF** operator you used in Chapter 3. To see why, consider this form:

```
(if x (format t "yes") (format t "no"))
```

If **IF** were a function, the evaluator would evaluate the argument expressions from left to right. The symbol `x` would be evaluated as a variable yielding some value; then `(format t "yes")` would be evaluated as a function call, yielding **NIL** after printing "yes" to standard output. Then `(format t "no")` would be evaluated, printing "no" and also yielding **NIL**. Only after all three expressions were evaluated would the resulting values be passed to **IF**, too late for it to control which of the two **FORMAT** expressions gets evaluated.

To solve this problem, Common Lisp defines a couple dozen so-called special operators, **IF** being one, that do things that functions can't do. There are 25 in all, but only a small handful are used directly in day-to-day programming.[13]

When the first element of a list is a symbol naming a special operator, the rest of the expressions are evaluated according to the rule for that operator.

The rule for **IF** is pretty easy: evaluate the first expression. If it evaluates to non-**NIL**, then evaluate the next expression and return its value. Otherwise, return the value of evaluating the third

expression or **NIL** if the third expression is omitted. In other words, the basic form of an **IF** expression is as follows:

```
(if test-form then-form [ else-form ])
```

The *test-form* will always be evaluated and then one or the other of the *then-form* or *else-form*.

An even simpler special operator is **QUOTE**, which takes a single expression as its "argument" and simply returns it, unevaluated. For instance, the following evaluates to the list `(+ 1 2)`, not the value 3:

```
(quote (+ 1 2))
```

There's nothing special about this list; you can manipulate it just like any list you could create with the **LIST** function.[14]

QUOTE is used commonly enough that a special syntax for it is built into the reader. Instead of writing the following:

```
(quote (+ 1 2))
```

you can write this:

```
'(+ 1 2)
```

This syntax is a small extension of the s-expression syntax understood by the reader. From the point of view of the evaluator, both those expressions will look the same: a list whose first element is the symbol **QUOTE** and whose second element is the list `(+ 1 2)`.[15]

In general, the special operators implement features of the language that require some special processing by the evaluator. For instance, several special operators manipulate the environment in which other

forms will be evaluated. One of these, which I'll discuss in detail in Chapter 6, is **LET**, which is used to create new variable bindings. The following form evaluates to 10 because the second `x` is evaluated in an environment where it's the name of a variable established by the **LET** with the value 10:

```
(let ((x 10)) x)
```

Macros

While special operators extend the syntax of Common Lisp beyond what can be expressed with just function calls, the set of special operators is fixed by the language standard. Macros, on the other hand, give users of the language a way to extend its syntax. As you saw in Chapter 3, a macro is a function that takes s-expressions as arguments and returns a Lisp form that's then evaluated in place of the macro form. The evaluation of a macro form proceeds in two phases: First, the elements of the macro form are passed, unevaluated, to the macro function. Second, the form returned by the macro function--called its *expansion*--is evaluated according to the normal evaluation rules.

It's important to keep the two phases of evaluating a macro form clear in your mind. It's easy to lose track when you're typing expressions at the REPL because the two phases happen one after another and the value of the second phase is immediately returned. But when Lisp code is compiled, the two phases happen at completely different times, so it's important to keep clear what's happening when. For instance, when you compile a whole file of source code with the function `COMPILE-FILE`, all the macro forms in the file are recursively expanded until the code consists of nothing but function call forms and special forms. This macroless code is then compiled into a FASL file that the `LOAD` function knows how to load. The compiled code, however, isn't executed until the file is loaded. Because macros generate their expansion at compile time, they can do relatively large amounts of work generating their

expansion without having to pay for it when the file is loaded or the functions defined in the file are called.

Since the evaluator doesn't evaluate the elements of the macro form before passing them to the macro function, they don't need to be well-formed Lisp forms. Each macro assigns a meaning to the s-expressions in the macro form by virtue of how it uses them to generate its expansion. In other words, each macro defines its own local syntax. For instance, the `backwards` macro from Chapter 3 defines a syntax in which an expression is a legal `backwards` form if it's a list that's the reverse of a legal Lisp form.

I'll talk quite a bit more about macros throughout this book. For now the important thing for you to realize is that macros--while syntactically similar to function calls--serve quite a different purpose, providing a hook into the compiler.[16]

Truth, Falsehood, and Equality

Two last bits of basic knowledge you need to get under your belt are Common Lisp's notion of truth and falsehood and what it means for two Lisp objects to be "equal." Truth and falsehood are--in this realm--straightforward: the symbol `NIL` is the only false value, and everything else is true. The symbol `T` is the canonical true value and can be used when you need to return a non-`NIL` value and don't have anything else handy. The only tricky thing about `NIL` is that it's the only object that's both an atom and a list: in addition to falsehood, it's also used to represent the empty list.[17] This equivalence between `NIL` and the empty list is built into the reader: if the reader sees `()`, it reads it as the symbol `NIL`. They're completely interchangeable. And because `NIL`, as I mentioned previously, is the name of a constant variable with the symbol `NIL` as its value, the expressions `nil`, `()`, `'nil`, and `'()` all evaluate to the same thing--the unquoted forms are evaluated as a reference to the constant variable whose value is the symbol `NIL`, but in the quoted forms the `QUOTE` special operator evaluates to the symbol directly. For the same reason, both `t` and `'t` will evaluate to the same thing: the symbol `T`.

Using phrases such as "the same thing" of course begs the question of what it means for two values to be "the same." As you'll see in future chapters, Common Lisp provides a number of type-specific equality predicates: `=` is used to compare numbers, `CHAR=` to compare characters, and so on. In this section I'll discuss the four "generic" equality predicates--functions that can be passed any two Lisp

objects and will return true if they're equivalent and false otherwise. They are, in order of discrimination, **EQ**, **EQL**, **EQUAL**, and **EQUALP**.

EQ tests for "object identity"--two objects are **EQ** if they're identical. Unfortunately, the object identity of numbers and characters depends on how those data types are implemented in a particular Lisp. Thus, **EQ** may consider two numbers or two characters with the same value to be equivalent, or it may not. Implementations have enough leeway that the expression `(eq 3 3)` can legally evaluate to either true or false. More to the point, `(eq x x)` can evaluate to either true or false if the value of `x` happens to be a number or character.

Thus, you should never use **EQ** to compare values that may be numbers or characters. It may seem to work in a predictable way for certain values in a particular implementation, but you have no guarantee that it will work the same way if you switch implementations. And switching implementations may mean simply upgrading your implementation to a new version--if your Lisp implementer changes how they represent numbers or characters, the behavior of **EQ** could very well change as well.

Thus, Common Lisp defines **EQL** to behave like **EQ** except that it also is guaranteed to consider two objects of the same class representing the same numeric or character value to be equivalent. Thus, `(eql 1 1)` is guaranteed to be true. And `(eql 1 1.0)` is guaranteed to be false since the integer value 1 and the floating-point value are instances of different classes.

There are two schools of thought about when to use **EQ** and when to use **EQL**: The "use **EQ** when possible" camp argues you should use **EQ**

when you know you aren't going to be com-paring numbers or characters because (a) it's a way to indicate that you aren't going to be comparing numbers or characters and (b) it will be marginally more efficient since EQ doesn't have to check whether its arguments are numbers or characters.

The "always use EQL" camp says you should never use EQ because (a) the potential gain in clarity is lost because every time someone reading your code--including you--sees an EQ, they have to stop and check whether it's being used correctly (in other words, that it's never going to be called upon to compare numbers or characters) and (b) that the efficiency difference between EQ and EQL is in the noise compared to real performance bottlenecks.

The code in this book is written in the "always use EQL" style.[18]

The other two equality predicates, EQUAL and EQUALP, are general in the sense that they can operate on all types of objects, but they're much less fundamental than EQ or EQL. They each define a slightly less discriminating notion of equivalence than EQL, allowing different objects to be considered equivalent. There's nothing special about the particular notions of equivalence these functions implement except that they've been found to be handy by Lisp programmers in the past. If these predicates don't suit your needs, you can always define your own predicate function that compares different types of objects in the way you need.

EQUAL loosens the discrimination of EQL to consider lists equivalent if they have the same structure and contents, recursively, according to EQUAL. EQUAL also considers strings equivalent if they contain the

same characters. It also defines a looser definition of equivalence than **EQL** for bit vectors and pathnames, two data types I'll discuss in future chapters. For all other types, it falls back on **EQL**.

EQUALP is similar to **EQUAL** except it's even less discriminating. It considers two strings equivalent if they contain the same characters, ignoring differences in case. It also considers two characters equivalent if they differ only in case. Numbers are equivalent under **EQUALP** if they represent the same mathematical value. Thus, `(equalp 1 1.0)` is true. Lists with **EQUALP** elements are **EQUALP**; likewise, arrays with **EQUALP** elements are **EQUALP**. As with **EQUAL**, there are a few other data types that I haven't covered yet for which **EQUALP** can consider two objects equivalent that neither **EQL** nor **EQUAL** will. For all other data types, **EQUALP** falls back on **EQL**.

Formatting Lisp Code

While code formatting is, strictly speaking, neither a syntactic nor a semantic matter, proper formatting is important to reading and writing code fluently and idiomatically. The key to formatting Lisp code is to indent it properly. The indentation should reflect the structure of the code so that you don't need to count parentheses to see what goes with what. In general, each new level of nesting gets indented a bit more, and, if line breaks are necessary, items at the same level of nesting are lined up. Thus, a function call that needs to be broken up across multiple lines might be written like this:

```
(some-function arg-with-a-long-name
               another-arg-with-an-even-longer-name)
```

Macro and special forms that implement control constructs are typically indented a little differently: the "body" elements are indented two spaces relative to the opening parenthesis of the form. Thus:

```
(defun print-list (list)
  (dolist (i list)
    (format t "item: ~a~%" i)))
```

However, you don't need to worry too much about these rules because a proper Lisp environment such as SLIME will take care of it for you. In fact, one of the advantages of Lisp's regular syntax is that it's fairly easy for software such as editors to know how to indent it. Since the indentation is supposed to reflect the structure of the code and the structure is marked by parentheses, it's easy to let the editor indent your code for you.

In SLIME, hitting Tab at the beginning of each line will cause it to be indented appropriately, or you can re-indent a whole expression by positioning the cursor on the opening parenthesis and typing `C-M-q`. Or you can re-indent the whole body of a function from anywhere within it by typing `C-c M-q`.

Indeed, experienced Lisp programmers tend to rely on their editor handling indenting automatically, not just to make their code look nice but to detect typos: once you get used to how code is supposed to be indented, a misplaced parenthesis will be instantly recognizable by the weird indentation your editor gives you. For example, suppose you were writing a function that was supposed to look like this:

```
(defun foo ()
  (if (test)
    (do-one-thing)
    (do-another-thing)))
```

Now suppose you accidentally left off the closing parenthesis after `test`. Because you don't bother counting parentheses, you quite likely would have added an extra parenthesis at the end of the **DEFUN** form, giving you this code:

```
(defun foo ()
  (if (test
    (do-one-thing)
    (do-another-thing))))
```

However, if you had been indenting by hitting Tab at the beginning of each line, you wouldn't have code like that. Instead you'd have this:

```
(defun foo ()
  (if (test
       (do-one-thing)
       (do-another-thing))))
```

Seeing the then and else clauses indented way out under the condition rather than just indented slightly relative to the **`IF`** shows you immediately that something is awry.

Another important formatting rule is that closing parentheses are always put on the same line as the last element of the list they're closing. That is, don't write this:

```
(defun foo ()
  (dotimes (i 10)
    (format t "~d. hello~%" i)
  )
)
```

but instead write this:

```
(defun foo ()
  (dotimes (i 10)
    (format t "~d. hello~%" i)))
```

The string of `)))`s at the end may seem forbidding, but as long your code is properly indented the parentheses should fade away--no need to give them undue prominence by spreading them across several lines.

Finally, comments should be prefaced with one to four semicolons depending on the scope of the comment as follows:

```
;;;; Four semicolons are used for a file header comment.

;;; A comment with three semicolons will usually be a paragraph
;;; comment that applies to a large section of code that follows,

(defun foo (x)
  (dotimes (i x)
    ;; Two semicolons indicate this comment applies to the code
    ;; that follows. Note that this comment is indented the same
    ;; as the code that follows.
    (some-function-call)
    (another i)              ; this comment applies to this line only
    (and-another)            ; and this is for this line
    (baz)))
```

Now you're ready to start looking in greater detail at the major building blocks of Lisp programs, functions, variables, and macros. Up next: functions.

---

[1] `http://www-formal.stanford.edu/jmc/history/lisp/node3.html`

[2] Lisp implementers, like implementers of any language, have many ways they can implement an evaluator, ranging from a "pure" interpreter that interprets the objects given to the evaluator directly to a compiler that translates the objects into machine code that it then runs. In the middle are implementations that compile the input into an intermediate form such as bytecodes for a virtual machine and then interprets the bytecodes. Most Common Lisp implementations these days use some form of compilation even when evaluating code at run time.

[3] Sometimes the phrase *s-expression* refers to the textual representation and sometimes to the objects that result from reading the textual representation. Usually either it's clear from context which is meant or the distinction isn't that important.

[4] Not all Lisp objects can be written out in a way that can be read back in. But anything you can **`READ`** can be printed back out "readably" with **`PRINT`**.

[5] The empty list, `()`, which can also be written **`NIL`**, is both an atom and a list.

[6] In fact, as you'll see later, names aren't intrinsically tied to any one kind of thing. You can use the same name, depending on context, to refer to both a variable and a function, not to mention several other possibilities.

[7] The case-converting behavior of the reader can, in fact, be customized, but understanding when and how to change it requires a much deeper discussion of the relation between names, symbols, and other program elements than I'm ready to get into just yet.

[8] I'll discuss the relation between symbols and packages in more detail in Chapter 21.

[9] Of course, other levels of correctness exist in Lisp, as in other languages. For instance, the s-expression that results from reading `(foo 1 2)` is syntactically well-formed but can be evaluated only if `foo` is the name of a function or macro.

[10]One other rarely used kind of Lisp form is a list whose first element is a *lambda form*. I'll discuss this kind of form in Chapter 5.

[11]One other possibility exists--it's possible to define *symbol macros* that are evaluated slightly differently. We won't worry about them.

[12]In Common Lisp a symbol can name both an operator--function, macro, or special operator--and a variable. This is one of the major differences between Common Lisp and Scheme. The difference is sometimes described as Common Lisp being a Lisp-2 vs. Scheme being a Lisp-1--a Lisp-2 has two namespaces, one for operators and one for variables, but a Lisp-1 uses a single namespace. Both choices have advantages, and partisans can debate endlessly which is better.

[13]The others provide useful, but somewhat esoteric, features. I'll discuss them as the features they support come up.

[14]Well, one difference exists--literal objects such as quoted lists, but also including double-quoted strings, literal arrays, and vectors (whose syntax you'll see later), must not be modified. Consequently, any lists you plan to manipulate you should create with `LIST`.

[15]This syntax is an example of a *reader macro*. Reader macros modify the syntax the reader uses to translate text into Lisp objects. It is, in fact, possible to define your own reader macros, but that's a rarely used facility of the language. When most Lispers talk about "extending the syntax" of the language, they're talking about regular macros, as I'll discuss in a moment.

[16]People without experience using Lisp's macros or, worse yet, bearing the scars of C preprocessor-inflicted wounds, tend to get nervous when they realize that macro calls look like regular function calls. This turns out not to be a problem in practice for several reasons. One is that macro forms are usually formatted differently than function calls. For instance, you write the following:

```
(dolist (x foo)
  (print x))
```

rather than this:

```
(dolist (x foo) (print x))
```

or

```
(dolist (x foo)
       (print x))
```

the way you would if DOLIST was a function. A good Lisp environment will automatically format macro calls correctly, even for user-defined macros.

And even if a DOLIST form was written on a single line, there are several clues that it's a macro: For one, the expression `(x foo)` is meaningful by itself only if `x` is the name of a function or macro. Combine that with the later occurrence of `x` as a variable, and it's pretty suggestive that DOLIST is a macro that's creating a binding for a variable named `x`. Naming conventions also help--looping constructs, which are invariably macros--are frequently given names starting with *do*.

[17]Using the empty list as false is a reflection of Lisp's heritage as a list-processing language much as the use of the integer 0 as false in C is a reflection of its heritage as a bit-twiddling language. Not all Lisps handle boolean values the same way. Another of the many subtle differences upon which a good Common Lisp vs. Scheme flame war can rage for days is Scheme's use of a distinct false value `#f`, which isn't the same value as either the symbol `nil` or the empty list, which are also distinct from each other.

[18]Even the language standard is a bit ambivalent about which of EQ or EQL should be preferred. *Object identity* is defined by EQ, but the standard defines the phrase *the same* when talking about objects to mean EQL unless another predicate is explicitly mentioned. Thus, if you want to be 100 percent technically correct, you can say that `(- 3 2)` and `(- 4 3)` evaluate to "the same" object but not that they evaluate to "identical" objects. This is, admittedly, a bit of an *angels-on-pinheads* kind of issue.

Copyright © 2003-2005, Peter Seibel

5. Functions

After the rules of syntax and semantics, the three most basic components of all Lisp programs are functions, variables and macros. You used all three while building the database in Chapter 3, but I glossed over a lot of the details of how they work and how to best use them. I'll devote the next few chapters to these three topics, starting with functions, which--like their counterparts in other languages--provide the basic mechanism for abstracting, well, functionality.

The bulk of Lisp itself consists of functions. More than three quarters of the names defined in the language standard name functions. All the built-in data types are defined purely in terms of what functions operate on them. Even Lisp's powerful object system is built upon a conceptual extension to functions, generic functions, which I'll cover in Chapter 16.

And, despite the importance of macros to The Lisp Way, in the end all real functionality is provided by functions. Macros run at compile time, so the code they generate--the code that will actually make up the program after all the macros are expanded--will consist entirely of calls to functions and special operators. Not to mention, macros themselves are also functions, albeit functions that are used to generate code rather than to perform the actions of the program.[1]

Defining New Functions

Normally functions are defined using the **DEFUN** macro. The basic skeleton of a **DEFUN** looks like this:

```
(defun name (parameter*)
  "Optional documentation string."
  body-form*)
```

Any symbol can be used as a function name.[2] Usually function names contain only alphabetic characters and hyphens, but other characters are allowed and are used in certain naming conventions. For instance, functions that convert one kind of value to another sometimes use `->` in the name. For example, a function to convert strings to widgets might be called `string->widget`. The most important naming convention is the one mentioned in Chapter 2, which is that you construct compound names with hyphens rather than underscores or inner caps. Thus, `frob-widget` is better Lisp style than either `frob_widget` or `frobWidget`.

A function's parameter list defines the variables that will be used to hold the arguments passed to the function when it's called.[3] If the function takes no arguments, the list is empty, written as `()`. Different flavors of parameters handle required, optional, multiple, and keyword arguments. I'll discuss the details in the next section.

If a string literal follows the parameter list, it's a documentation string that should describe the purpose of the function. When the function is defined, the documentation string will be associated with the name of the function and can later be obtained using the **DOCUMENTATION** function.[4]

Finally, the body of a DEFUN consists of any number of Lisp expressions. They will be evaluated in order when the function is called and the value of the last expression is returned as the value of the function. Or the RETURN-FROM special operator can be used to return immediately from anywhere in a function, as I'll discuss in a moment.

In Chapter 2 we wrote a `hello-world` function, which looked like this:

```
(defun hello-world () (format t "hello, world"))
```

You can now analyze the parts of this function. Its name is `hello-world`, its parameter list is empty so it takes no arguments, it has no documentation string, and its body consists of one expression.

```
(format t "hello, world")
```

The following is a slightly more complex function:

```
(defun verbose-sum (x y)
  "Sum any two numbers after printing a message."
  (format t "Summing ~d and ~d.~%" x y)
  (+ x y))
```

This function is named `verbose-sum`, takes two arguments that will be bound to the parameters `x` and `y`, has a documentation string, and has a body consisting of two expressions. The value returned by the call to + becomes the return value of `verbose-sum`.

Function Parameter Lists

There's not a lot more to say about function names or documentation strings, and it will take a good portion of the rest of this book to describe all the things you can do in the body of a function, which leaves us with the parameter list.

The basic purpose of a parameter list is, of course, to declare the variables that will receive the arguments passed to the function. When a parameter list is a simple list of variable names--as in `verbose-sum`--the parameters are called *required parameters*. When a function is called, it must be supplied with one argument for every required parameter. Each parameter is bound to the corresponding argument. If a function is called with too few or too many arguments, Lisp will signal an error.

However, Common Lisp's parameter lists also give you more flexible ways of mapping the arguments in a function call to the function's parameters. In addition to required parameters, a function can have optional parameters. Or a function can have a single parameter that's bound to a list containing any extra arguments. And, finally, arguments can be mapped to parameters using keywords rather than position. Thus, Common Lisp's parameter lists provide a convenient solution to several common coding problems.

Optional Parameters

While many functions, like `verbose-sum`, need only required parameters, not all functions are quite so simple. Sometimes a function will have a parameter that only certain callers will care about, perhaps because there's a reasonable default value. An example is a function that creates a data structure that can grow as needed. Since the data structure can grow, it doesn't matter--from a correctness point of view--what the initial size is. But callers who have a good idea how many items they're going to put into the data structure may be able to improve performance by specifying a specific initial size. Most callers, though, would probably rather let the code that implements the data structure pick a good general-purpose value. In Common Lisp you can accommodate both kinds of callers by using an optional parameter; callers who don't care will get a reasonable default, and other callers can provide a specific value.[5]

To define a function with optional parameters, after the names of any required parameters, place the symbol **`&optional`** followed by the names of the optional parameters. A simple example looks like this:

```
(defun foo (a b &optional c d) (list a b c d))
```

When the function is called, arguments are first bound to the required parameters. After all the required parameters have been given values, if there are any arguments left, their values are assigned to the optional parameters. If the arguments run out before the optional parameters do, the remaining optional parameters are

bound to the value **NIL**. Thus, the function defined previously gives the following results:

```
(foo 1 2)     ==> (1 2 NIL NIL)
(foo 1 2 3)   ==> (1 2 3 NIL)
(foo 1 2 3 4) ==> (1 2 3 4)
```

Lisp will still check that an appropriate number of arguments are passed to the function--in this case between two and four, inclusive--and will signal an error if the function is called with too few or too many.

Of course, you'll often want a different default value than **NIL**. You can specify the default value by replacing the parameter name with a list containing a name and an expression. The expression will be evaluated only if the caller doesn't pass enough arguments to provide a value for the optional parameter. The common case is simply to provide a value as the expression.

```
(defun foo (a &optional (b 10)) (list a b))
```

This function requires one argument that will be bound to the parameter `a`. The second parameter, `b`, will take either the value of the second argument, if there is one, or 10.

```
(foo 1 2) ==> (1 2)
(foo 1)   ==> (1 10)
```

Sometimes, however, you may need more flexibility in choosing the default value. You may want to compute a default value based on other parameters. And you can--the default-value expression can refer to parameters that occur earlier in the parameter list. If you were writing a function that returned some sort of representation of a

rectangle and you wanted to make it especially convenient to make squares, you might use an argument list like this:

```
(defun make-rectangle (width &optional (height width)) ...)
```

which would cause the `height` parameter to take the same value as the `width` parameter unless explicitly specified.

Occasionally, it's useful to know whether the value of an optional argument was supplied by the caller or is the default value. Rather than writing code to check whether the value of the parameter is the default (which doesn't work anyway, if the caller happens to explicitly pass the default value), you can add another variable name to the parameter specifier after the default-value expression. This variable will be bound to true if the caller actually supplied an argument for this parameter and **NIL** otherwise. By convention, these variables are usually named the same as the actual parameter with a "-supplied-p" on the end. For example:

```
(defun foo (a b &optional (c 3 c-supplied-p))
  (list a b c c-supplied-p))
```

This gives results like this:

```
(foo 1 2)   ==> (1 2 3 NIL)
(foo 1 2 3) ==> (1 2 3 T)
(foo 1 2 4) ==> (1 2 4 T)
```

Rest Parameters

Optional parameters are just the thing when you have discrete parameters for which the caller may or may not want to provide values. But some functions need to take a variable number of arguments. Several of the built-in functions you've seen already work this way. `FORMAT` has two required arguments, the stream and the control string. But after that it needs a variable number of arguments depending on how many values need to be interpolated into the control string. The `+` function also takes a variable number of arguments--there's no particular reason to limit it to summing just two numbers; it will sum any number of values. (It even works with zero arguments, returning 0, the identity under addition.) The following are all legal calls of those two functions:

```
(format t "hello, world")
(format t "hello, ~a" name)
(format t "x: ~d y: ~d" x y)
(+)
(+ 1)
(+ 1 2)
(+ 1 2 3)
```

Obviously, you could write functions taking a variable number of arguments by simply giving them a lot of optional parameters. But that would be incredibly painful--just writing the parameter list would be bad enough, and that doesn't get into dealing with all the parameters in the body of the function. To do it properly, you'd have to have as many optional parameters as the number of arguments that can legally be passed in a function call. This number is implementation dependent but guaranteed to be at least 50. And in

current implementations it ranges from 4,096 to 536,870,911.[6] Blech. That kind of mind-bending tedium is definitely *not* The Lisp Way.

Instead, Lisp lets you include a catchall parameter after the symbol **`&rest`**. If a function includes a **`&rest`** parameter, any arguments remaining after values have been doled out to all the required and optional parameters are gathered up into a list that becomes the value of the **`&rest`** parameter. Thus, the parameter lists for **`FORMAT`** and **`+`** probably look something like this:

```
(defun format (stream string &rest values) ...)
(defun + (&rest numbers) ...)
```

Keyword Parameters

Optional and rest parameters give you quite a bit of flexibility, but neither is going to help you out much in the following situation: Suppose you have a function that takes four optional parameters. Now suppose that most of the places the function is called, the caller wants to provide a value for only one of the four parameters and, further, that the callers are evenly divided as to which parameter they will use.

The callers who want to provide a value for the first parameter are fine--they just pass the one optional argument and leave off the rest. But all the other callers have to pass some value for between one and three arguments they don't care about. Isn't that exactly the problem optional parameters were designed to solve?

Of course it is. The problem is that optional parameters are still positional--if the caller wants to pass an explicit value for the fourth optional parameter, it turns the first three optional parameters into required parameters for that caller. Luckily, another parameter flavor, keyword parameters, allow the caller to specify which values go with which parameters.

To give a function keyword parameters, after any required, **`&optional`**, and **`&rest`** parameters you include the symbol **`&key`** and then any number of keyword parameter specifiers, which work like optional parameter specifiers. Here's a function that has only keyword parameters:

```
(defun foo (&key a b c) (list a b c))
```

When this function is called, each keyword parameters is bound to the value immediately following a keyword of the same name. Recall from Chapter 4 that keywords are names that start with a colon and that they're automatically defined as self-evaluating constants.

If a given keyword doesn't appear in the argument list, then the corresponding parameter is assigned its default value, just like an optional parameter. Because the keyword arguments are labeled, they can be passed in any order as long as they follow any required arguments. For instance, `foo` can be invoked as follows:

```
(foo)                 ==> (NIL NIL NIL)
(foo :a 1)            ==> (1 NIL NIL)
(foo :b 1)            ==> (NIL 1 NIL)
(foo :c 1)            ==> (NIL NIL 1)
(foo :a 1 :c 3)       ==> (1 NIL 3)
(foo :a 1 :b 2 :c 3) ==> (1 2 3)
(foo :a 1 :c 3 :b 2) ==> (1 2 3)
```

As with optional parameters, keyword parameters can provide a default value form and the name of a supplied-p variable. In both keyword and optional parameters, the default value form can refer to parameters that appear earlier in the parameter list.

```
(defun foo (&key (a 0) (b 0 b-supplied-p) (c (+ a b)))
  (list a b c b-supplied-p))

(foo :a 1)            ==> (1 0 1 NIL)
(foo :b 1)            ==> (0 1 1 T)
(foo :b 1 :c 4)       ==> (0 1 4 T)
(foo :a 2 :b 1 :c 4) ==> (2 1 4 T)
```

Also, if for some reason you want the keyword the caller uses to specify the parameter to be different from the name of the actual parameter, you can replace the parameter name with another list containing the keyword to use when calling the function and the name to be used for the parameter. The following definition of `foo`:

```
(defun foo (&key ((:apple a)) ((:box b) 0) ((:charlie c) 0 c-supplied-p))
  (list a b c c-supplied-p))
```

lets the caller call it like this:

```
(foo :apple 10 :box 20 :charlie 30) ==> (10 20 30 T)
```

This style is mostly useful if you want to completely decouple the public API of the function from the internal details, usually because you want to use short variable names internally but descriptive keywords in the API. It's not, however, very frequently used.

Mixing Different Parameter Types

It's possible, but rare, to use all four flavors of parameters in a single function. Whenever more than one flavor of parameter is used, they must be declared in the order I've discussed them: first the names of the required parameters, then the optional parameters, then the rest parameter, and finally the keyword parameters. Typically, however, in functions that use multiple flavors of parameters, you'll combine required parameters with one other flavor or possibly combine **`&optional`** and **`&rest`** parameters. The other two combinations, either **`&optional`** or **`&rest`** parameters combined with **`&key`** parameters, can lead to somewhat surprising behavior.

Combining **`&optional`** and **`&key`** parameters yields surprising enough results that you should probably avoid it altogether. The problem is that if a caller doesn't supply values for all the optional parameters, then those parameters will eat up the keywords and values intended for the keyword parameters. For instance, this function unwisely mixes **`&optional`** and **`&key`** parameters:

```
(defun foo (x &optional y &key z) (list x y z))
```

If called like this, it works fine:

```
(foo 1 2 :z 3) ==> (1 2 3)
```

And this is also fine:

```
(foo 1)  ==> (1 nil nil)
```

But this will signal an error:

```
(foo 1 :z 3) ==> ERROR
```

This is because the keyword `:z` is taken as a value to fill the optional `y` parameter, leaving only the argument 3 to be processed. At that point, Lisp will be expecting either a keyword/value pair or nothing and will complain. Perhaps even worse, if the function had had two **`&optional`** parameters, this last call would have resulted in the values `:z` and 3 being bound to the two **`&optional`** parameters and the **`&key`** parameter `z` getting the default value **`NIL`** with no indication that anything was amiss.

In general, if you find yourself writing a function that uses both **`&optional`** and **`&key`** parameters, you should probably just change it to use all **`&key`** parameters--they're more flexible, and you can always add new keyword parameters without disturbing existing callers of the function. You can also remove keyword parameters, as long as no one is using them.[7] In general, using keyword parameters helps make code much easier to maintain and evolve--if you need to add some new behavior to a function that requires new parameters, you can add keyword parameters without having to touch, or even recompile, any existing code that calls the function.

You can safely combine **`&rest`** and **`&key`** parameters, but the behavior may be a bit surprising initially. Normally the presence of either **`&rest`** or **`&key`** in a parameter list causes all the values remaining after the required and **`&optional`** parameters have been filled in to be processed in a particular way--either gathered into a list for a **`&rest`** parameter or assigned to the appropriate **`&key`** parameters based on the keywords. If both **`&rest`** and **`&key`** appear in a parameter list, then both things happen--all the remaining values, which include the keywords themselves, are gathered into a list that's bound to the

&rest parameter, and the appropriate values are also bound to the **&key** parameters. So, given this function:

```
(defun foo (&rest rest &key a b c) (list rest a b c))
```

you get this result:

```
(foo :a 1 :b 2 :c 3)  ==> ((:A 1 :B 2 :C 3) 1 2 3)
```

Function Return Values

All the functions you've written so far have used the default behavior of returning the value of the last expression evaluated as their own return value. This is the most common way to return a value from a function.

However, sometimes it's convenient to be able to return from the middle of a function such as when you want to break out of nested control constructs. In such cases you can use the `RETURN-FROM` special operator to immediately return any value from the function.

You'll see in Chapter 20 that `RETURN-FROM` is actually not tied to functions at all; it's used to return from a block of code defined with the `BLOCK` special operator. However, `DEFUN` automatically wraps the whole function body in a block with the same name as the function. So, evaluating a `RETURN-FROM` with the name of the function and the value you want to return will cause the function to immediately exit with that value. `RETURN-FROM` is a special operator whose first "argument" is the name of the block from which to return. This name isn't evaluated and thus isn't quoted.

The following function uses nested loops to find the first pair of numbers, each less than 10, whose product is greater than the argument, and it uses `RETURN-FROM` to return the pair as soon as it finds it:

```
(defun foo (n)
  (dotimes (i 10)
    (dotimes (j 10)
      (when (> (* i j) n)
        (return-from foo (list i j))))))
```

Admittedly, having to specify the name of the function you're returning from is a bit of a pain--for one thing, if you change the function's name, you'll need to change the name used in the `RETURN-FROM` as well.[8] But it's also the case that explicit `RETURN-FROM`s are used much less frequently in Lisp than `return` statements in C-derived languages, because *all* Lisp expressions, including control constructs such as loops and conditionals, evaluate to a value. So it's not much of a problem in practice.

Functions As Data, a.k.a. Higher-Order Functions

While the main way you use functions is to call them by name, a number of situations exist where it's useful to be able treat functions as data. For instance, if you can pass one function as an argument to another, you can write a general-purpose sorting function while allowing the caller to provide a function that's responsible for comparing any two elements. Then the same underlying algorithm can be used with many different comparison functions. Similarly, callbacks and hooks depend on being able to store references to code in order to run it later. Since functions are already the standard way to abstract bits of code, it makes sense to allow functions to be treated as data.[9]

In Lisp, functions are just another kind of object. When you define a function with `DEFUN`, you're really doing two things: creating a new function object and giving it a name. It's also possible, as you saw in Chapter 3, to use `LAMBDA` expressions to create a function without giving it a name. The actual representation of a function object, whether named or anonymous, is opaque--in a native-compiling Lisp, it probably consists mostly of machine code. The only things you need to know are how to get hold of it and how to invoke it once you've got it.

The special operator `FUNCTION` provides the mechanism for getting at a function object. It takes a single argument and returns the function with that name. The name isn't quoted. Thus, if you've defined a function `foo`, like so:

```
CL-USER> (defun foo (x) (* 2 x))
FOO
```

you can get the function object like this:[10]

```
CL-USER> (function foo)
#<Interpreted Function FOO>
```

In fact, you've already used **FUNCTION**, but it was in disguise. The syntax `#'`, which you used in Chapter 3, is syntactic sugar for **FUNCTION**, just the way `'` is syntactic sugar for **QUOTE**.[11] Thus, you can also get the function object for `foo` like this:

```
CL-USER> #'foo
#<Interpreted Function FOO>
```

Once you've got the function object, there's really only one thing you can do with it--invoke it. Common Lisp provides two functions for invoking a function through a function object: **FUNCALL** and **APPLY**.[12] They differ only in how they obtain the arguments to pass to the function.

FUNCALL is the one to use when you know the number of arguments you're going to pass to the function at the time you write the code. The first argument to **FUNCALL** is the function object to be invoked, and the rest of the arguments are passed onto that function. Thus, the following two expressions are equivalent:

```
(foo 1 2 3) === (funcall #'foo 1 2 3)
```

However, there's little point in using **FUNCALL** to call a function whose name you know when you write the code. In fact, the previous two expressions will quite likely compile to exactly the same machine instructions.

The following function demonstrates a more apt use of **FUNCALL**. It accepts a function object as an argument and plots a simple ASCII-art histogram of the values returned by the argument function when it's invoked on the values from `min` to `max`, stepping by `step`.

```
(defun plot (fn min max step)
  (loop for i from min to max by step do
        (loop repeat (funcall fn i) do (format t "*"))
        (format t "~%")))
```

The **FUNCALL** expression computes the value of the function for each value of `i`. The inner **LOOP** uses that computed value to determine how many times to print an asterisk to standard output.

Note that you don't use **FUNCTION** or `#'` to get the function value of `fn`; you *want* it to be interpreted as a variable because it's the variable's value that will be the function object. You can call `plot` with any function that takes a single numeric argument, such as the built-in function **EXP** that returns the value of *e* raised to the power of its argument.

```
CL-USER> (plot #'exp 0 4 1/2)
*
*
**
****
*******
************
********************
*********************************
***************************************************
NIL
```

FUNCALL, however, doesn't do you any good when the argument list is known only at runtime. For instance, to stick with the `plot` function for another moment, suppose you've obtained a list containing a function object, a minimum and maximum value, and a step value. In other words, the list contains the values you want to pass as

arguments to `plot`. Suppose this list is in the variable `plot-data`. You could invoke `plot` on the values in that list like this:

```
(plot (first plot-data) (second plot-data) (third plot-data) (fourth plot-data))
```

This works fine, but it's pretty annoying to have to explicitly unpack the arguments just so you can pass them to `plot`.

That's where **`APPLY`** comes in. Like **`FUNCALL`**, the first argument to **`APPLY`** is a function object. But after the function object, instead of individual arguments, it expects a list. It then applies the function to the values in the list. This allows you to write the following instead:

```
(apply #'plot plot-data)
```

As a further convenience, **`APPLY`** can also accept "loose" arguments as long as the last argument is a list. Thus, if `plot-data` contained just the min, max, and step values, you could still use **`APPLY`** like this to plot the **`EXP`** function over that range:

```
(apply #'plot #'exp plot-data)
```

`APPLY` doesn't care about whether the function being applied takes **`&optional`**, **`&rest`**, or **`&key`** arguments--the argument list produced by combining any loose arguments with the final list must be a legal argument list for the function with enough arguments for all the required parameters and only appropriate keyword parameters.

Anonymous Functions

Once you start writing, or even simply using, functions that accept other functions as arguments, you're bound to discover that sometimes it's annoying to have to define and name a whole separate function that's used in only one place, especially when you never call it by name.

When it seems like overkill to define a new function with **DEFUN**, you can create an "anonymous" function using a **LAMBDA** expression. As discussed in Chapter 3, a **LAMBDA** expression looks like this:

```
(lambda (parameters) body)
```

One way to think of **LAMBDA** expressions is as a special kind of function name where the name itself directly describes what the function does. This explains why you can use a **LAMBDA** expression in the place of a function name with `#'`.

```
(funcall #'(lambda (x y) (+ x y)) 2 3) ==> 5
```

You can even use a **LAMBDA** expression as the "name" of a function in a function call expression. If you wanted, you could write the previous **FUNCALL** expression more concisely.

```
((lambda (x y) (+ x y)) 2 3) ==> 5
```

But this is almost never done; it's merely worth noting that it's legal in order to emphasize that **LAMBDA** expressions can be used anywhere a normal function name can be.[13]

Anonymous functions can be useful when you need to pass a function as an argument to another function and the function you need to pass is simple enough to express inline. For instance, suppose you wanted to plot the function *2x*. You *could* define the following function:

```
(defun double (x) (* 2 x))
```

which you could then pass to `plot`.

```
CL-USER> (plot #'double 0 10 1)

**
****
******
********
**********
************
**************
****************
******************
********************
NIL
```

But it's easier, and arguably clearer, to write this:

```
CL-USER> (plot #'(lambda (x) (* 2 x)) 0 10 1)

**
****
******
********
**********
************
**************
****************
******************
********************
NIL
```

The other important use of **`LAMBDA`** expressions is in making *closures*, functions that capture part of the environment where they're created. You used closures a bit in Chapter 3, but the details of how closures work and what they're used for is really more about how variables work than functions, so I'll save that discussion for the next chapter.

[1]Despite the importance of functions in Common Lisp, it isn't really accurate to describe it as a *functional* language. It's true some of Common Lisp's features, such as its list manipulation functions, are designed to be used in a body-form* style and that Lisp has a prominent place in the history of functional programming--McCarthy introduced many ideas that are now considered important in functional programming--but Common Lisp was intentionally designed to support many different styles of programming. In the Lisp family, Scheme is the nearest thing to a "pure" functional language, and even it has several features that disqualify it from absolute purity compared to languages such as Haskell and ML.

[2]Well, almost any symbol. It's undefined what happens if you use any of the names defined in the language standard as a name for one of your own functions. However, as you'll see in Chapter 21, the Lisp package system allows you to create names in different namespaces, so this isn't really an issue.

[3]Parameter lists are sometimes also called *lambda lists* because of the historical relationship between Lisp's notion of functions and the lambda calculus.

[4]For example, the following:

```
(documentation 'foo 'function)
```

returns the documentation string for the function `foo`. Note, however, that documentation strings are intended for human consumption, not programmatic access. A Lisp implementation isn't *required* to store them and is allowed to discard them at any time, so portable programs shouldn't depend on their presence. In some implementations an implementation-defined variable needs to be set before it will store documentation strings.

[5]In languages that don't support optional parameters directly, programmers typically find ways to simulate them. One technique is to use distinguished "no-value" values that the caller can pass to indicate they want the default value of a given parameter. In C, for example, it's common to use `NULL` as such a distinguished value. However, such a protocol between the function and its callers is ad hoc--in some functions or for some arguments `NULL` may be the distinguished value while in other functions or for other arguments the magic value may be -1 or some `#defined` constant.

[6]The constant **`CALL-ARGUMENTS-LIMIT`** tells you the implementation-specific value.

[7]Four standard functions take both **`&optional`** and **`&key`** arguments--**`READ-FROM-STRING`**, **`PARSE-NAMESTRING`**, **`WRITE-LINE`**, and **`WRITE-STRING`**. They were left that way during

standardization for backward compatibility with earlier Lisp dialects. READ-FROM-STRING tends to be the one that catches new Lisp programmers most frequently--a call such as `(read-from-string s :start 10)` seems to ignore the `:start` keyword argument, reading from index 0 instead of 10. That's because READ-FROM-STRING also has two &optional parameters that swallowed up the arguments `:start` and 10.

[8]Another macro, RETURN, doesn't require a name. However, you can't use it instead of RETURN-FROM to avoid having to specify the function name; it's syntactic sugar for returning from a block named NIL. I'll cover it, along with the details of BLOCK and RETURN-FROM, in Chapter 20.

[9]Lisp, of course, isn't the only language to treat functions as data. C uses function pointers, Perl uses subroutine references, Python uses a scheme similar to Lisp, and C# introduces delegates, essentially typed function pointers, as an improvement over Java's rather clunky reflection and anonymous class mechanisms.

[10]The exact printed representation of a function object will differ from implementation to implementation.

[11]The best way to think of FUNCTION is as a special kind of quotation. QUOTEing a symbol prevents it from being evaluated at all, resulting in the symbol itself rather than the value of the variable named by that symbol. FUNCTION also circumvents the normal evaluation rule but, instead of preventing the symbol from being evaluated at all, causes it to be evaluated as the name of a function, just the way it would if it were used as the function name in a function call expression.

[12]There's actually a third, the special operator MULTIPLE-VALUE-CALL, but I'll save that for when I discuss expressions that return multiple values in Chapter 20.

[13]In Common Lisp it's also possible to use a LAMBDA expression as an argument to FUNCALL (or some other function that takes a function argument such as SORT or MAPCAR) with no `#'` before it, like this:

```
(funcall (lambda (x y) (+ x y)) 2 3)
```

This is legal and is equivalent to the version with the `#'` but for a tricky reason. Historically LAMBDA expressions by themselves weren't expressions that could be evaluated. That is LAMBDA wasn't the name of a function, macro, or special operator. Rather, a list starting with the symbol LAMBDA was a special syntactic construct that Lisp recognized as a kind of function name.

But if that were still true, then `(funcall (lambda (...) ...))` would be illegal because **FUNCALL** is a function and the normal evaluation rule for a function call would require that the **LAMBDA** expression be evaluated. However, late in the ANSI standardization process, in order to make it possible to implement ISLISP, another Lisp dialect being standardized at the same time, strictly as a user-level compatibility layer on top of Common Lisp, a **LAMBDA** macro was defined that expands into a call to **FUNCTION** wrapped around the **LAMBDA** expression. In other words, the following **LAMBDA** expression:

```
(lambda () 42)
```

exands into the following when it occurs in a context where it evaluated:

```
(function (lambda () 42))   ; or #'(lambda () 42)
```

This makes its use in a value position, such as an argument to **FUNCALL**, legal. In other words, it's pure syntactic sugar. Most folks either always use `#'` before **LAMBDA** expressions in value positions or never do. In this book, I always use `#'`.

Copyright © 2003-2005, Peter Seibel

6. Variables

The next basic building block we need to look at are variables. Common Lisp supports two kinds of variables: lexical and dynamic.[1] These two types correspond roughly to "local" and "global" variables in other languages. However, the correspondence is only approximate. On one hand, some languages' "local" variables are in fact much like Common Lisp's dynamic variables.[2] And on the other, some languages' local variables are *lexically scoped* without providing all the capabilities provided by Common Lisp's lexical variables. In particular, not all languages that provide lexically scoped variables support closures.

To make matters a bit more confusing, many of the forms that deal with variables can be used with both lexical and dynamic variables. So I'll start by discussing a few aspects of Lisp's variables that apply to both kinds and then cover the specific characteristics of lexical and dynamic variables. Then I'll discuss Common Lisp's general-purpose assignment operator, `SETF`, which is used to assign new values to variables and just about every other place that can hold a value.

Variable Basics

As in other languages, in Common Lisp variables are named places that can hold a value. However, in Common Lisp, variables aren't typed the way they are in languages such as Java or C++. That is, you don't need to declare the type of object that each variable can hold. Instead, a variable can hold values of any type and the values carry type information that can be used to check types at runtime. Thus, Common Lisp is *dynamically typed*--type errors are detected dynamically. For instance, if you pass something other than a number to the `+` function, Common Lisp will signal a type error. On the other hand, Common Lisp *is* a *strongly typed* language in the sense that all type errors will be detected--there's no way to treat an object as an instance of a class that it's not.[3]

All values in Common Lisp are, conceptually at least, references to objects.[4] Consequently, assigning a variable a new value changes *what* object the variable refers to but has no effect on the previously referenced object. However, if a variable holds a reference to a mutable object, you can use that reference to modify the object, and the modification will be visible to any code that has a reference to the same object.

One way to introduce new variables you've already used is to define function parameters. As you saw in the previous chapter, when you define a function with `DEFUN`, the parameter list defines the variables that will hold the function's arguments when it's called. For example, this function defines three variables--`x`, `y`, and `z`--to hold its arguments.

```
(defun foo (x y z) (+ x y z))
```

Each time a function is called, Lisp creates new *bindings* to hold the arguments passed by the function's caller. A binding is the runtime manifestation of a variable. A single variable--the thing you can point to in the program's source code--can have many different bindings during a run of the program. A single variable can even have multiple bindings at the same time; parameters to a recursive function, for example, are rebound for each call to the function.

As with all Common Lisp variables, function parameters hold object references.[5] Thus, you can assign a new value to a function parameter within the body of the function, and it will not affect the bindings created for another call to the same function. But if the object passed to a function is mutable and you change it in the function, the changes will be visible to the caller since both the caller and the callee will be referencing the same object.

Another form that introduces new variables is the **LET** special operator. The skeleton of a **LET** form looks like this:

```
(let (variable*)
  body-form*)
```

where each *variable* is a variable initialization form. Each initialization form is either a list containing a variable name and an initial value form or--as a shorthand for initializing the variable to **NIL**--a plain variable name. The following **LET** form, for example, binds the three variables `x`, `y`, and `z` with initial values 10, 20, and **NIL**:

```
(let ((x 10) (y 20) z)
  ...)
```

When the **LET** form is evaluated, all the initial value forms are first evaluated. Then new bindings are created and initialized to the appropriate initial values before the body forms are executed. Within the body of the **LET**, the variable names refer to the newly created bindings. After the **LET**, the names refer to whatever, if anything, they referred to before the **LET**.

The value of the last expression in the body is returned as the value of the **LET** expression. Like function parameters, variables introduced with **LET** are rebound each time the **LET** is entered.[6]

The *scope* of function parameters and **LET** variables--the area of the program where the variable name can be used to refer to the variable's binding--is delimited by the form that introduces the variable. This form--the function definition or the **LET**--is called the *binding form*. As you'll see in a bit, the two types of variables--lexical and dynamic--use two slightly different scoping mechanisms, but in both cases the scope is delimited by the binding form.

If you nest binding forms that introduce variables with the same name, then the bindings of the innermost variable *shadows* the outer bindings. For instance, when the following function is called, a binding is created for the parameter `x` to hold the function's argument. Then the first **LET** creates a new binding with the initial value 2, and the inner **LET** creates yet another binding, this one with the initial value 3. The bars on the right mark the scope of each binding.

```
(defun foo (x)
  (format t "Parameter: ~a~%" x)      ; |<------ x is argument
  (let ((x 2))                        ; |
    (format t "Outer LET: ~a~%" x)    ; | |<---- x is 2
```

```
    (let ((x 3))                        ; | |
      (format t "Inner LET: ~a~%" x)) ; | | |<-- x is 3
    (format t "Outer LET: ~a~%" x))   ; | |
  (format t "Parameter: ~a~%" x))     ; |
```

Each reference to `x` will refer to the binding with the smallest enclosing scope. Once control leaves the scope of one binding form, the binding from the immediately enclosing scope is unshadowed and `x` refers to it instead. Thus, calling `foo` results in this output:

```
CL-USER> (foo 1)
Parameter: 1
Outer LET: 2
Inner LET: 3
Outer LET: 2
Parameter: 1
NIL
```

In future chapters I'll discuss other constructs that also serve as binding forms--any construct that introduces a new variable name that's usable only within the construct is a binding form.

For instance, in Chapter 7 you'll meet the **DOTIMES** loop, a basic counting loop. It introduces a variable that holds the value of a counter that's incremented each time through the loop. The following loop, for example, which prints the numbers from 0 to 9, binds the variable `x`:

```
(dotimes (x 10) (format t "~d " x))
```

Another binding form is a variant of **LET**, **LET***. The difference is that in a **LET**, the variable names can be used only in the body of the **LET**--the part of the **LET** after the variables list--but in a **LET***, the initial value forms for each variable can refer to variables introduced earlier in the variables list. Thus, you can write the following:

```
(let* ((x 10)
       (y (+ x 10)))
  (list x y))
```

but not this:

```
(let ((x 10)
      (y (+ x 10)))
  (list x y))
```

However, you could achieve the same result with nested **LET**s.

```
(let ((x 10))
  (let ((y (+ x 10)))
    (list x y)))
```

Lexical Variables and Closures

By default all binding forms in Common Lisp introduce *lexically scoped* variables. Lexically scoped variables can be referred to only by code that's textually within the binding form. Lexical scoping should be familiar to anyone who has programmed in Java, C, Perl, or Python since they all provide lexically scoped "local" variables. For that matter, Algol programmers should also feel right at home, as Algol first introduced lexical scoping in the 1960s.

However, Common Lisp's lexical variables are lexical variables with a twist, at least compared to the original Algol model. The twist is provided by the combination of lexical scoping with nested functions. By the rules of lexical scoping, only code textually within the binding form can refer to a lexical variable. But what happens when an anonymous function contains a reference to a lexical variable from an enclosing scope? For instance, in this expression:

```
(let ((count 0)) #'(lambda () (setf count (1+ count))))
```

the reference to `count` inside the **LAMBDA** form should be legal according to the rules of lexical scoping. Yet the anonymous function containing the reference will be returned as the value of the **LET** form and can be invoked, via **FUNCALL**, by code that's *not* in the scope of the **LET**. So what happens? As it turns out, when `count` is a lexical variable, it just works. The binding of `count` created when the flow of control entered the **LET** form will stick around for as long as needed, in this case for as long as someone holds onto a reference to the function object returned by the **LET** form. The anonymous function is

called a *closure* because it "closes over" the binding created by the **LET**.

The key thing to understand about closures is that it's the binding, not the value of the variable, that's captured. Thus, a closure can not only access the value of the variables it closes over but can also assign new values that will persist between calls to the closure. For instance, you can capture the closure created by the previous expression in a global variable like this:

```
(defparameter *fn* (let ((count 0)) #'(lambda () (setf count (1+ count)))))
```

Then each time you invoke it, the value of count will increase by one.

```
CL-USER> (funcall *fn*)
1
CL-USER> (funcall *fn*)
2
CL-USER> (funcall *fn*)
3
```

A single closure can close over many variable bindings simply by referring to them. Or multiple closures can capture the same binding. For instance, the following expression returns a list of three closures, one that increments the value of the closed over `count` binding, one that decrements it, and one that returns the current value:

```
(let ((count 0))
  (list
   #'(lambda () (incf count))
   #'(lambda () (decf count))
   #'(lambda () count)))
```

Dynamic, a.k.a. Special, Variables

Lexically scoped bindings help keep code understandable by limiting the scope, literally, in which a given name has meaning. This is why most modern languages use lexical scoping for local variables. Sometimes, however, you really want a global variable--a variable that you can refer to from anywhere in your program. While it's true that indiscriminate use of global variables can turn code into spaghetti nearly as quickly as unrestrained use of `goto`, global variables do have legitimate uses and exist in one form or another in almost every programming language.[7] And as you'll see in a moment, Lisp's version of global variables, dynamic variables, are both more useful and more manageable.

Common Lisp provides two ways to create global variables: **DEFVAR** and **DEFPARAMETER**. Both forms take a variable name, an initial value, and an optional documentation string. After it has been **DEFVAR**ed or **DEFPARAMETER**ed, the name can be used anywhere to refer to the current binding of the global variable. As you've seen in previous chapters, global variables are conventionally named with names that start and end with `*`. You'll see later in this section why it's quite important to follow that naming convention. Examples of **DEFVAR** and **DEFPARAMETER** look like this:

```
(defvar *count* 0
  "Count of widgets made so far.")

(defparameter *gap-tolerance* 0.001
  "Tolerance to be allowed in widget gaps.")
```

The difference between the two forms is that **DEFPARAMETER** always assigns the initial value to the named variable while **DEFVAR** does so

only if the variable is undefined. A **DEFVAR** form can also be used with no initial value to define a global variable without giving it a value. Such a variable is said to be *unbound*.

Practically speaking, you should use **DEFVAR** to define variables that will contain data you'd want to keep even if you made a change to the source code that uses the variable. For instance, suppose the two variables defined previously are part of an application for controlling a widget factory. It's appropriate to define the `*count*` variable with **DEFVAR** because the number of widgets made so far isn't invalidated just because you make some changes to the widget-making code.[8]

On the other hand, the variable `*gap-tolerance*` presumably has some effect on the behavior of the widget-making code itself. If you decide you need a tighter or looser tolerance and change the value in the **DEFPARAMETER** form, you'd like the change to take effect when you recompile and reload the file.

After defining a variable with **DEFVAR** or **DEFPARAMETER**, you can refer to it from anywhere. For instance, you might define this function to increment the count of widgets made:

```
(defun increment-widget-count () (incf *count*))
```

The advantage of global variables is that you don't have to pass them around. Most languages store the standard input and output streams in global variables for exactly this reason--you never know when you're going to want to print something to standard out, and you don't want every function to have to accept and pass on

arguments containing those streams just in case someone further down the line needs them.

However, once a value, such as the standard output stream, is stored in a global variable and you have written code that references that global variable, it's tempting to try to temporarily modify the behavior of that code by changing the variable's value.

For instance, suppose you're working on a program that contains some low-level logging functions that print to the stream in the global variable `*standard-output*`. Now suppose that in part of the program you want to capture all the output generated by those functions into a file. You might open a file and assign the resulting stream to `*standard-output*`. Now the low-level functions will send their output to the file.

This works fine until you forget to set `*standard-output*` back to the original stream when you're done. If you forget to reset `*standard-output*`, all the other code in the program that uses `*standard-output*` will also send its output to the file.[9]

What you really want, it seems, is a way to wrap a piece of code in something that says, "All code below here--all the functions it calls, all the functions they call, and so on, down to the lowest-level functions--should use *this* value for the global variable `*standard-output*`." Then when the high-level function returns, the old value of `*standard-output*` should be automatically restored.

It turns out that that's exactly what Common Lisp's other kind of variable--dynamic variables--let you do. When you bind a dynamic

variable--for example, with a **LET** variable or a function parameter--the binding that's created on entry to the binding form replaces the global binding for the duration of the binding form. Unlike a lexical binding, which can be referenced by code only within the lexical scope of the binding form, a dynamic binding can be referenced by any code that's invoked during the execution of the binding form.[10] And it turns out that all global variables are, in fact, dynamic variables.

Thus, if you want to temporarily redefine `*standard-output*`, the way to do it is simply to rebind it, say, with a **LET**.

```
(let ((*standard-output* *some-other-stream*))
  (stuff))
```

In any code that runs as a result of the call to `stuff`, references to `*standard-output*` will use the binding established by the **LET**. And when `stuff` returns and control leaves the **LET**, the new binding of `*standard-output*` will go away and subsequent references to `*standard-output*` will see the binding that was current before the **LET**. At any given time, the most recently established binding shadows all other bindings. Conceptually, each new binding for a given dynamic variable is pushed onto a stack of bindings for that variable, and references to the variable always use the most recent binding. As binding forms return, the bindings they created are popped off the stack, exposing previous bindings.[11]

A simple example shows how this works.

```
(defvar *x* 10)
(defun foo () (format t "X: ~d~%" *x*))
```

The **DEFVAR** creates a global binding for the variable `*x*` with the value 10. The reference to `*x*` in `foo` will look up the current binding dynamically. If you call `foo` from the top level, the global binding created by the **DEFVAR** is the only binding available, so it prints 10.

```
CL-USER> (foo)
X: 10
NIL
```

But you can use **LET** to create a new binding that temporarily shadows the global binding, and `foo` will print a different value.

```
CL-USER> (let ((*x* 20)) (foo))
X: 20
NIL
```

Now call `foo` again, with no **LET**, and it again sees the global binding.

```
CL-USER> (foo)
X: 10
NIL
```

Now define another function.

```
(defun bar ()
  (foo)
  (let ((*x* 20)) (foo))
  (foo))
```

Note that the middle call to `foo` is wrapped in a **LET** that binds `*x*` to the new value 20. When you run `bar`, you get this result:

```
CL-USER> (bar)
X: 10
X: 20
X: 10
NIL
```

As you can see, the first call to `foo` sees the global binding, with its value of 10. The middle call, however, sees the new binding, with the value 20. But after the **LET**, `foo` once again sees the global binding.

As with lexical bindings, assigning a new value affects only the current binding. To see this, you can redefine `foo` to include an assignment to `*x*`.

```
(defun foo ()
  (format t "Before assignment~18tX: ~d~%" *x*)
  (setf *x* (+ 1 *x*))
  (format t "After assignment~18tX: ~d~%" *x*))
```

Now `foo` prints the value of `*x*`, increments it, and prints it again. If you just run `foo`, you'll see this:

```
CL-USER> (foo)
Before assignment X: 10
After assignment  X: 11
NIL
```

Not too surprising. Now run `bar`.

```
CL-USER> (bar)
Before assignment X: 11
After assignment  X: 12
Before assignment X: 20
After assignment  X: 21
Before assignment X: 12
After assignment  X: 13
NIL
```

Notice that `*x*` started at 11--the earlier call to `foo` really did change the global value. The first call to `foo` from `bar` increments the global binding to 12. The middle call doesn't see the global binding because of the **`LET`**. Then the last call can see the global binding again and increments it from 12 to 13.

So how does this work? How does **`LET`** know that when it binds `*x*` it's supposed to create a dynamic binding rather than a normal lexical binding? It knows because the name has been declared *special*.[12] The name of every variable defined with **`DEFVAR`** and **`DEFPARAMETER`** is automatically declared globally special. This means

whenever you use such a name in a binding form--in a **LET** or as a function parameter or any other construct that creates a new variable binding--the binding that's created will be a dynamic binding. This is why the `*naming* *convention*` is so important--it'd be bad news if you used a name for what you thought was a lexical variable and that variable happened to be globally special. On the one hand, code you call could change the value of the binding out from under you; on the other, you might be shadowing a binding established by code higher up on the stack. If you always name global variables according to the `*` naming convention, you'll never accidentally use a dynamic binding where you intend to establish a lexical binding.

It's also possible to declare a name locally special. If, in a binding form, you declare a name special, then the binding created for that variable will be dynamic rather than lexical. Other code can locally declare a name special in order to refer to the dynamic binding. However, locally special variables are relatively rare, so you needn't worry about them.[13]

Dynamic bindings make global variables much more manageable, but it's important to notice they still allow action at a distance. Binding a global variable has two at a distance effects--it can change the behavior of downstream code, and it also opens the possibility that downstream code will assign a new value to a binding established higher up on the stack. You should use dynamic variables only when you need to take advantage of one or both of these characteristics.

Constants

One other kind of variable I haven't mentioned at all is the oxymoronic "constant variable." All constants are global and are defined with **DEFCONSTANT**. The basic form of **DEFCONSTANT** is like **DEFPARAMETER**.

```
(defconstant name initial-value-form [ documentation-string ])
```

As with **DEFVAR** and **DEFPARAMETER**, **DEFCONSTANT** has a global effect on the name used--thereafter the name can be used only to refer to the constant; it can't be used as a function parameter or rebound with any other binding form. Thus, many Lisp programmers follow a naming convention of using names starting and ending with `+` for constants. This convention is somewhat less universally followed than the `*`-naming convention for globally special names but is a good idea for the same reason.[14]

Another thing to note about **DEFCONSTANT** is that while the language allows you to redefine a constant by reevaluating a **DEFCONSTANT** with a different initial-value-form, what exactly happens after the redefinition isn't defined. In practice, most implementations will require you to reevaluate any code that refers to the constant in order to see the new value since the old value may well have been inlined. Consequently, it's a good idea to use **DEFCONSTANT** only to define things that are *really* constant, such as the value of NIL. For things you might ever want to change, you should use **DEFPARAMETER** instead.

Assignment

Once you've created a binding, you can do two things with it: get the current value and set it to a new value. As you saw in Chapter 4, a symbol evaluates to the value of the variable it names, so you can get the current value simply by referring to the variable. To assign a new value to a binding, you use the **SETF** macro, Common Lisp's general-purpose assignment operator. The basic form of **SETF** is as follows:

```
(setf place value)
```

Because **SETF** is a macro, it can examine the form of the *place* it's assigning to and expand into appropriate lower-level operations to manipulate that place. When the place is a variable, it expands into a call to the special operator **SETQ**, which, as a special operator, has access to both lexical and dynamic bindings.[15] For instance, to assign the value 10 to the variable `x`, you can write this:

```
(setf x 10)
```

As I discussed earlier, assigning a new value to a binding has no effect on any other bindings of that variable. And it doesn't have any effect on the value that was stored in the binding prior to the assignment. Thus, the **SETF** in this function:

```
(defun foo (x) (setf x 10))
```

will have no effect on any value outside of `foo`. The binding that was created when `foo` was called is set to 10, immediately replacing

whatever value was passed as an argument. In particular, a form such as the following:

```
(let ((y 20))
  (foo y)
  (print y))
```

will print 20, not 10, as it's the value of `y` that's passed to `foo` where it's briefly the value of the variable `x` before the **`SETF`** gives `x` a new value.

`SETF` can also assign to multiple places in sequence. For instance, instead of the following:

```
(setf x 1)
(setf y 2)
```

you can write this:

```
(setf x 1 y 2)
```

`SETF` returns the newly assigned value, so you can also nest calls to **`SETF`** as in the following expression, which assigns both `x` and `y` the same random value:

```
(setf x (setf y (random 10)))
```

Generalized Assignment

Variable bindings, of course, aren't the only places that can hold values. Common Lisp supports composite data structures such as arrays, hash tables, and lists, as well as user-defined data structures, all of which consist of multiple places that can each hold a value.

I'll cover those data structures in future chapters, but while we're on the topic of assignment, you should note that **SETF** can assign any place a value. As I cover the different composite data structures, I'll point out which functions can serve as "**SETF**able places." The short version, however, is if you need to assign a value to a place, **SETF** is almost certainly the tool to use. It's even possible to extend **SETF** to allow it to assign to user-defined places though I won't cover that.[16]

In this regard **SETF** is no different from the = assignment operator in most C-derived languages. In those languages, the = operator assigns new values to variables, array elements, and fields of classes. In languages such as Perl and Python that support hash tables as a built-in data type, = can also set the values of individual hash table entries. Table 6-1 summarizes the various ways = is used in those languages.

Table 6-1. Assignment with = in Other Languages

Assigning to ...	Java, C, C++	Perl	Python
... variable	`x = 10;`	`$x = 10;`	`x = 10`
... array element	`a[0] = 10;`	`$a[0] = 10;`	`a[0] = 10`
... hash table	--	`$hash{'key'} = 10;`	`hash['key'] = 10`

entry	
... field in object	`o.field = 10; $o->{'field'} = 10; o.field = 10`

SETF works the same way--the first "argument" to **SETF** is a place to store the value, and the second argument provides the value. As with the = operator in these languages, you use the same form to express the place as you'd normally use to fetch the value.[17] Thus, the Lisp equivalents of the assignments in Table 6-1--given that **AREF** is the array access function, **GETHASH** does a hash table lookup, and `field` might be a function that accesses a slot named `field` of a user-defined object--are as follows:

```
Simple variable:    (setf x 10)
Array:              (setf (aref a 0) 10)
Hash table:         (setf (gethash 'key hash) 10)
Slot named 'field': (setf (field o) 10)
```

Note that **SETF**ing a place that's part of a larger object has the same semantics as **SETF**ing a variable: the place is modified without any effect on the object that was previously stored in the place. Again, this is similar to how = behaves in Java, Perl, and Python.[18]

Other Ways to Modify Places

While all assignments can be expressed with **SETF**, certain patterns involving assigning a new value based on the current value are sufficiently common to warrant their own operators. For instance, while you could increment a number with **SETF**, like this:

```
(setf x (+ x 1))
```

or decrement it with this:

```
(setf x (- x 1))
```

it's a bit tedious, compared to the C-style `++x` and `--x`. Instead, you can use the macros **INCF** and **DECF**, which increment and decrement a place by a certain amount that defaults to 1.

```
(incf x)    === (setf x (+ x 1))
(decf x)    === (setf x (- x 1))
(incf x 10) === (setf x (+ x 10))
```

INCF and **DECF** are examples of a kind of macro called *modify macros*. Modify macros are macros built on top of **SETF** that modify places by assigning a new value based on the current value of the place. The main benefit of modify macros is that they're more concise than the same modification written out using **SETF**. Additionally, modify macros are defined in a way that makes them safe to use with places where the place expression must be evaluated only once. A silly example is this expression, which increments the value of an arbitrary element of an array:

```
(incf (aref *array* (random (length *array*))))
```

A naive translation of that into a **SETF** expression might look like this:

```
(setf (aref *array* (random (length *array*)))
      (1+ (aref *array* (random (length *array*)))))
```

However, that doesn't work because the two calls to **RANDOM** won't necessarily return the same value--this expression will likely grab the value of one element of the array, increment it, and then store it back as the new value of a different element. The **INCF** expression, however, does the right thing because it knows how to take apart this expression:

```
(aref *array* (random (length *array*)))
```

to pull out the parts that could possibly have side effects to make sure they're evaluated only once. In this case, it would probably expand into something more or less equivalent to this:

```
(let ((tmp (random (length *array*))))
  (setf (aref *array* tmp) (1+ (aref *array* tmp))))
```

In general, modify macros are guaranteed to evaluate both their arguments and the subforms of the place form exactly once each, in left-to-right order.

The macro **PUSH**, which you used in the mini-database to add elements to the `*db*` variable, is another modify macro. You'll take a closer look at how it and its counterparts **POP** and **PUSHNEW** work in Chapter 12 when I talk about how lists are represented in Lisp.

Finally, two slightly esoteric but useful modify macros are **ROTATEF** and **SHIFTF**. **ROTATEF** rotates values between places. For instance, if you have two variables, `a` and `b`, this call:

```
(rotatef a b)
```

swaps the values of the two variables and returns **NIL**. Since `a` and `b` are variables and you don't have to worry about side effects, the previous **ROTATEF** expression is equivalent to this:

```
(let ((tmp a)) (setf a b b tmp) nil)
```

With other kinds of places, the equivalent expression using **SETF** would be quite a bit more complex.

SHIFTF is similar except instead of rotating values it shifts them to the left--the last argument provides a value that's moved to the second-to-last argument while the rest of the values are moved one to the left. The original value of the first argument is simply returned. Thus, the following:

```
(shiftf a b 10)
```

is equivalent--again, since you don't have to worry about side effects--to this:

```
(let ((tmp a)) (setf a b b 10) tmp)
```

Both **ROTATEF** and **SHIFTF** can be used with any number of arguments and, like all modify macros, are guaranteed to evaluate them exactly once, in left to right order.

With the basics of Common Lisp's functions and variables under your belt, now you're ready to move onto the feature that continues to differentiate Lisp from other languages: macros.

---

[1] Dynamic variables are also sometimes called *special variables* for reasons you'll see later in this chapter. It's important to be aware of this synonym, as some folks (and Lisp implementations) use one term while others use the other.

[2]Early Lisps tended to use dynamic variables for local variables, at least when interpreted. Elisp, the Lisp dialect used in Emacs, is a bit of a throwback in this respect, continuing to support only dynamic variables. Other languages have recapitulated this transition from dynamic to lexical variables--Perl's `local` variables, for instance, are dynamic while its `my` variables, introduced in Perl 5, are lexical. Python never had true dynamic variables but only introduced true lexical scoping in version 2.2. (Python's lexical variables are still somewhat limited compared to Lisp's because of the conflation of assignment and binding in the language's syntax.)

[3]Actually, it's not quite true to say that all type errors will always be detected--it's possible to use optional declarations to tell the compiler that certain variables will always contain objects of a particular type and to turn off runtime type checking in certain regions of code. However, declarations of this sort are used to optimize code after it has been developed and debugged, not during normal development.

[4]As an optimization certain kinds of objects, such as integers below a certain size and characters, may be represented directly in memory where other objects would be represented by a pointer to the actual object. However, since integers and characters are immutable, it doesn't matter that there may be multiple copies of "the same" object in different variables. This is the root of the difference between **EQ** and **EQL** discussed in Chapter 4.

[5]In compiler-writer terms Common Lisp functions are "pass-by-value." However, the values that are passed are references to objects. This is similar to how Java and Python work.

[6]The variables in **LET** forms and function parameters are created by exactly the same mechanism. In fact, in some Lisp dialects--though not Common Lisp--**LET** is simply a macro that expands into a call to an anonymous function. That is, in those dialects, the following:

```
(let ((x 10)) (format t "~a" x))
```

is a macro form that expands into this:

```
((lambda (x) (format t "~a" x)) 10)
```

[7]Java disguises global variables as public static fields, C uses `extern` variables, and Python's module-level and Perl's package-level variables can likewise be accessed from anywhere.

[8]If you specifically want to reset a **DEFVAR**ed variable, you can either set it directly with **SETF** or make it unbound using **MAKUNBOUND** and then reevaluate the **DEFVAR** form.

[9]The strategy of temporarily reassigning *standard-output* also breaks if the system is multithreaded--if there are multiple threads of control trying to print to different streams at the same time, they'll all try to set the global variable to the stream they want to use, stomping all over each other. You could use a lock to control access to the global variable, but then you're not really getting the benefit of multiple concurrent threads, since whatever thread is printing has to lock out all the other threads until it's done even if they want to print to a different stream.

[10]The technical term for the interval during which references may be made to a binding is its *extent*. Thus, *scope* and *extent* are complementary notions--scope refers to space while extent refers to time. Lexical variables have lexical scope but *indefinite* extent, meaning they stick around for an indefinite interval, determined by how long they're needed. Dynamic variables, by contrast, have indefinite scope since they can be referred to from anywhere but *dynamic* extent. To further confuse matters, the combination of indefinite scope and dynamic extent is frequently referred to by the misnomer *dynamic scope*.

[11]Though the standard doesn't specify how to incorporate multithreading into Common Lisp, implementations that provide multithreading follow the practice established on the Lisp machines and create dynamic bindings on a per-thread basis. A reference to a global variable will find the binding most recently established in the current thread, or the global binding.

[12]This is why dynamic variables are also sometimes called *special variables*.

[13]If you must know, you can look up `DECLARE`, `SPECIAL`, and `LOCALLY` in the HyperSpec.

[14]Several key constants defined by the language itself don't follow this convention--not least of which are `T` and `NIL`. This is occasionally annoying when one wants to use `t` as a local variable name. Another is `PI`, which holds the best long-float approximation of the mathematical constant pi.

[15]Some old-school Lispers prefer to use `SETQ` with variables, but modern style tends to use `SETF` for all assignments.

[16]Look up `DEFSETF`, `DEFINE-SETF-EXPANDER` for more information.

[17]The prevalence of Algol-derived syntax for assignment with the "place" on the left side of the `=` and the new value on the right side has spawned the terminology *lvalue*, short for "left value," meaning something that can be assigned to, and *rvalue*, meaning something that provides a value. A compiler hacker would say, "`SETF` treats its first argument as an lvalue."

[18]C programmers may want to think of variables and other places as holding a pointer to the real object; assigning to a variable simply changes what object it points to while assigning to a part of a composite object is similar to indirecting through the pointer to the actual object. C++ programmers should note that the behavior of = in C++ when dealing with objects--namely, a memberwise copy--is quite idiosyncratic.

Copyright © 2003-2005, Peter Seibel

7. Macros: Standard Control Constructs

While many of the ideas that originated in Lisp, from the conditional expression to garbage collection, have been incorporated into other languages, the one language feature that continues to set Common Lisp apart is its macro system. Unfortunately, the word *macro* describes a lot of things in computing to which Common Lisp's macros bear only a vague and metaphorical similarity. This causes no end of misunderstanding when Lispers try to explain to non-Lispers what a great feature macros are.[1] To understand Lisp's macros, you really need to come at them fresh, without preconceptions based on other things that also happen to be called macros. So let's start our discussion of Lisp's macros by taking a step back and looking at various ways languages support extensibility.

All programmers should be used to the idea that the definition of a language can include a standard library of functionality that's implemented in terms of the "core" language--functionality that could have been implemented by any programmer on top of the language if it hadn't been defined as part of the standard library. C's standard library, for instance, can be implemented almost entirely in portable C. Similarly, most of the ever-growing set of classes and interfaces that ship with Java's standard Java Development Kit (JDK) are written in "pure" Java.

One advantage of defining languages in terms of a core plus a standard library is it makes them easier to understand and implement. But the real benefit is in terms of expressiveness--since much of what you think of as "the language" is really just a library--the language is easy to extend. If C doesn't have a function to do some thing or another that you need, you can write that function, and now you have a slightly richer version of C. Similarly, in a language such as Java or Smalltalk where almost all the interesting parts of the "language" are defined in terms of classes, by defining new classes you extend the language, making it more suited for writing programs to do whatever it is you're trying to do.

While Common Lisp supports both these methods of extending the language, macros give Common Lisp yet another way. As I discussed briefly in Chapter 4, each macro defines its own syntax, determining how the s-expressions it's passed are turned into Lisp forms. With macros as part of the core language it's possible to build new syntax--control constructs such as `WHEN`, `DOLIST`, and `LOOP` as well as definitional forms such as `DEFUN` and `DEFPARAMETER`--as part of the "standard library" rather than having to hardwire them into the core. This has implications for how the language itself is implemented, but as a Lisp programmer you'll care more that it gives you another way to extend the language, making it a better language for expressing solutions to your particular programming problems.

Now, it may seem that the benefits of having another way to extend the language would be easy to recognize. But for some reason a lot of folks who haven't actually used Lisp macros--folks who think nothing of spending their days creating new functional abstractions

or defining hierarchies of classes to solve their programming problems--get spooked by the idea of being able to define new syntactic abstractions. The most common cause of macrophobia seems to be bad experiences with other "macro" systems. Simple fear of the unknown no doubt plays a role, too. To avoid triggering any macrophobic reactions, I'll ease into the subject by discussing several of the standard control-construct macros defined by Common Lisp. These are some of the things that, if Lisp didn't have macros, would have to be built into the language core. When you use them, you don't have to care that they're implemented as macros, but they provide a good example of some of the things you can do with macros.[2] In the next chapter, I'll show you how you can define your own macros.

WHEN and UNLESS

As you've already seen, the most basic form of conditional execution--if *x*, do *y*; otherwise do *z*--is provided by the **IF** special operator, which has this basic form:

```
(if condition then-form [else-form])
```

The *condition* is evaluated and, if its value is non-**NIL**, the *then-form* is evaluated and the resulting value returned. Otherwise, the *else-form*, if any, is evaluated and its value returned. If *condition* is **NIL** and there's no *else-form*, then the **IF** returns **NIL**.

```
(if (> 2 3) "Yup" "Nope") ==> "Nope"
(if (> 2 3) "Yup")        ==> NIL
(if (> 3 2) "Yup" "Nope") ==> "Yup"
```

However, **IF** isn't actually such a great syntactic construct because the *then-form* and *else-form* are each restricted to being a single Lisp form. This means if you want to perform a sequence of actions in either clause, you need to wrap them in some other syntax. For instance, suppose in the middle of a spam-filtering program you wanted to both file a message as spam and update the spam database when a message is spam. You can't write this:

```
(if (spam-p current-message)
    (file-in-spam-folder current-message)
    (update-spam-database current-message))
```

because the call to `update-spam-database` will be treated as the else clause, not as part of the then clause. Another special operator, **PROGN**, executes any number of forms in order and returns the value of the last form. So you could get the desired behavior by writing the following:

```
(if (spam-p current-message)
    (progn
      (file-in-spam-folder current-message)
      (update-spam-database current-message)))
```

That's not too horrible. But given the number of times you'll likely have to use this idiom, it's not hard to imagine that you'd get tired of it after a while. "Why," you might ask yourself, "doesn't Lisp provide a way to say what I really want, namely, 'When *x* is true, do this, that, and the other thing'?" In other words, after a while you'd notice the pattern of an **IF** plus a **PROGN** and wish for a way to abstract away the details rather than writing them out every time.

This is exactly what macros provide. In this case, Common Lisp comes with a standard macro, **WHEN**, which lets you write this:

```
(when (spam-p current-message)
  (file-in-spam-folder current-message)
  (update-spam-database current-message))
```

But if it wasn't built into the standard library, you could define **WHEN** yourself with a macro such as this, using the backquote notation I discussed in Chapter 3:[3]

```
(defmacro when (condition &rest body)
  `(if ,condition (progn ,@body)))
```

A counterpart to the **WHEN** macro is **UNLESS**, which reverses the condition, evaluating its body forms only if the condition is false. In other words:

```
(defmacro unless (condition &rest body)
  `(if (not ,condition) (progn ,@body)))
```

Admittedly, these are pretty trivial macros. There's no deep black magic here; they just abstract away a few language-level bookkeeping details, allowing you to express your true intent a bit

more clearly. But their very triviality makes an important point: because the macro system is built right into the language, you can write trivial macros like `WHEN` and `UNLESS` that give you small but real gains in clarity that are then multiplied by the thousands of times you use them. In Chapters 24, 26, and 31 you'll see how macros can also be used on a larger scale, creating whole domain-specific embedded languages. But first let's finish our discussion of the standard control-construct macros.

COND

Another time raw `IF` expressions can get ugly is when you have a multibranch conditional: if *a* do *x*, else if *b* do *y*; else do *z*. There's no logical problem writing such a chain of conditional expressions with just `IF`, but it's not pretty.

```
(if a
    (do-x)
    (if b
       (do-y)
       (do-z)))
```

And it would be even worse if you needed to include multiple forms in the then clauses, requiring `PROGN`s. So, not surprisingly, Common Lisp provides a macro for expressing multibranch conditionals: `COND`. This is the basic skeleton:

```
(cond
  (test-1 form*)
      .
      .
      .
  (test-N form*))
```

Each element of the body represents one branch of the conditional and consists of a list containing a condition form and zero or more forms to be evaluated if that branch is chosen. The conditions are evaluated in the order the branches appear in the body until one of them evaluates to true. At that point, the remaining forms in that branch are evaluated, and the value of the last form in the branch is returned as the value of the `COND` as a whole. If the branch contains no forms after the condition, the value of the condition is returned instead. By convention, the branch representing the final else clause in an if/else-if chain is written with a condition of `T`. Any non-`NIL` value

will work, but a `T` serves as a useful landmark when reading the code. Thus, you can write the previous nested `IF` expression using `COND` like this:

```
(cond (a (do-x))
      (b (do-y))
      (t (do-z)))
```

AND, OR, and NOT

When writing the conditions in IF, WHEN, UNLESS, and COND forms, three operators that will come in handy are the boolean logic operators, AND, OR, and NOT.

NOT is a function so strictly speaking doesn't belong in this chapter, but it's closely tied to AND and OR. It takes a single argument and inverts its truth value, returning T if the argument is NIL and NIL otherwise.

AND and OR, however, are macros. They implement logical conjunction and disjunction of any number of subforms and are defined as macros so they can *short-circuit*. That is, they evaluate only as many of their subforms--in left-to-right order--as necessary to determine the overall truth value. Thus, AND stops and returns NIL as soon as one of its subforms evaluates to NIL. If all the subforms evaluate to non-NIL, it returns the value of the last subform. OR, on the other hand, stops as soon as one of its subforms evaluates to non-NIL and returns the resulting value. If none of the subforms evaluate to true, OR returns NIL. Here are some examples:

```
(not nil)             ==> T
(not (= 1 1))         ==> NIL
(and (= 1 2) (= 3 3)) ==> NIL
(or (= 1 2) (= 3 3))  ==> T
```

Looping

Control constructs are the other main kind of looping constructs. Common Lisp's looping facilities are--in addition to being quite powerful and flexible--an interesting lesson in the have-your-cake-and-eat-it-too style of programming that macros provide.

As it turns out, none of Lisp's 25 special operators directly support structured looping. All of Lisp's looping control constructs are macros built on top of a pair of special operators that provide a primitive goto facility.[4] Like many good abstractions, syntactic or otherwise, Lisp's looping macros are built as a set of layered abstractions starting from the base provided by those two special operators.

At the bottom (leaving aside the special operators) is a very general looping construct, `DO`. While very powerful, `DO` suffers, as do many general-purpose abstractions, from being overkill for simple situations. So Lisp also provides two other macros, `DOLIST` and `DOTIMES`, that are less flexible than `DO` but provide convenient support for the common cases of looping over the elements of a list and counting loops. While an implementation can implement these macros however it wants, they're typically implemented as macros that expand into an equivalent `DO` loop. Thus, `DO` provides a basic structured looping construct on top of the underlying primitives provided by Common Lisp's special operators, and `DOLIST` and `DOTIMES` provide two easier-to-use, if less general, constructs. And, as you'll see in the next chapter, you can build your own looping constructs on top of `DO` for situations where `DOLIST` and `DOTIMES` don't meet your needs.

Finally, the `LOOP` macro provides a full-blown mini-language for expressing looping constructs in a non-Lispy, English-like (or at least Algol-like) language. Some Lisp hackers love `LOOP`; others hate it. `LOOP`'s fans like it because it provides a concise way to express certain commonly needed looping constructs. Its detractors dislike it because it's not Lispy enough. But whichever side one comes down on, it's a remarkable example of the power of macros to add new constructs to the language.

DOLIST and DOTIMES

I'll start with the easy-to-use **DOLIST** and **DOTIMES** macros.

DOLIST loops across the items of a list, executing the loop body with a variable holding the successive items of the list.[5] This is the basic skeleton (leaving out some of the more esoteric options):

```
(dolist (var list-form)
  body-form*)
```

When the loop starts, the *list-form* is evaluated once to produce a list. Then the body of the loop is evaluated once for each item in the list with the variable *var* holding the value of the item. For instance:

```
CL-USER> (dolist (x '(1 2 3)) (print x))
1
2
3
NIL
```

Used this way, the **DOLIST** form as a whole evaluates to **NIL**.

If you want to break out of a **DOLIST** loop before the end of the list, you can use **RETURN**.

```
CL-USER> (dolist (x '(1 2 3)) (print x) (if (evenp x) (return)))
1
2
NIL
```

DOTIMES is the high-level looping construct for counting loops. The basic template is much the same as **DOLIST**'s.

```
(dotimes (var count-form)
  body-form*)
```

The *count-form* must evaluate to an integer. Each time through the loop *var* holds successive integers from 0 to one less than that number. For instance:

```
CL-USER> (dotimes (i 4) (print i))
0
1
2
3
NIL
```

As with **DOLIST**, you can use **RETURN** to break out of the loop early.

Because the body of both **DOLIST** and **DOTIMES** loops can contain any kind of expressions, you can also nest loops. For example, to print out the times tables from `1 × 1 = 1` to `20 × 20 = 400`, you can write this pair of nested **DOTIMES** loops:

```
(dotimes (x 20)
  (dotimes (y 20)
    (format t "~3d " (* (1+ x) (1+ y))))
  (format t "~%"))
```

DO

While DOLIST and DOTIMES are convenient and easy to use, they aren't flexible enough to use for all loops. For instance, what if you want to step multiple variables in parallel? Or use an arbitrary expression to test for the end of the loop? If neither DOLIST nor DOTIMES meet your needs, you still have access to the more general DO loop.

Where DOLIST and DOTIMES provide only one loop variable, DO lets you bind any number of variables and gives you complete control over how they change on each step through the loop. You also get to define the test that determines when to end the loop and can provide a form to evaluate at the end of the loop to generate a return value for the DO expression as a whole. The basic template looks like this:

```
(do (variable-definition*)
    (end-test-form result-form*)
  statement*)
```

Each *variable-definition* introduces a variable that will be in scope in the body of the loop. The full form of a single variable definition is a list containing three elements.

```
(var init-form step-form)
```

The *init-form* will be evaluated at the beginning of the loop and the resulting values bound to the variable *var*. Before each subsequent iteration of the loop, the *step-form* will be evaluated and the new value assigned to *var*. The *step-form* is optional; if it's left out, the variable will keep its value from iteration to iteration unless you explicitly assign it a new value in the loop body. As with the variable

definitions in a LET, if the *init-form* is left out, the variable is bound to NIL. Also as with LET, you can use a plain variable name as shorthand for a list containing just the name.

At the beginning of each iteration, after all the loop variables have been given their new values, the *end-test-form* is evaluated. As long as it evaluates to NIL, the iteration proceeds, evaluating the *statements* in order.

When the *end-test-form* evaluates to true, the *result-forms* are evaluated, and the value of the last result form is returned as the value of the DO expression.

At each step of the iteration the step forms for all the variables are evaluated before assigning any of the values to the variables. This means you can refer to any of the other loop variables in the step forms.[6] That is, in a loop like this:

```
(do ((n 0 (1+ n))
     (cur 0 next)
     (next 1 (+ cur next)))
    ((= 10 n) cur))
```

the step forms `(1+ n)`, `next`, and `(+ cur next)` are all evaluated using the old values of `n`, `cur`, and `next`. Only after all the step forms have been evaluated are the variables given their new values. (Mathematically inclined readers may notice that this is a particularly efficient way of computing the eleventh Fibonacci number.)

This example also illustrates another characteristic of DO--because you can step multiple variables, you often don't need a body at all. Other times, you may leave out the result form, particularly if you're

just using the loop as a control construct. This flexibility, however, is the reason that DO expressions can be a bit cryptic. Where exactly do all the parentheses go? The best way to understand a DO expression is to keep in mind the basic template.

```
(do (variable-definition*)
    (end-test-form result-form*)
  statement*)
```

The six parentheses in that template are the only ones required by the DO itself. You need one pair to enclose the variable declarations, one pair to enclose the end test and result forms, and one pair to enclose the whole expression. Other forms within the DO may require their own parentheses--variable definitions are usually lists, for instance. And the test form is often a function call. But the skeleton of a DO loop will always be the same. Here are some example DO loops with the skeleton in bold:

```
(do ((i 0 (1+ i)))
    ((>= i 4))
  (print i))
```

Notice that the result form has been omitted. This is, however, not a particularly idiomatic use of DO, as this loop is much more simply written using DOTIMES.[7]

```
(dotimes (i 4) (print i))
```

As another example, here's the bodiless Fibonacci-computing loop:

```
(do ((n 0 (1+ n))
     (cur 0 next)
     (next 1 (+ cur next)))
    ((= 10 n) cur))
```

Finally, the next loop demonstrates a DO loop that binds no variables. It loops while the current time is less than the value of a global

variable, printing "Waiting" once a minute. Note that even with no loop variables, you still need the empty variables list.

```
(do ()
    ((> (get-universal-time) *some-future-date*))
  (format t "Waiting~%")
  (sleep 60))
```

The Mighty LOOP

For the simple cases you have DOLIST and DOTIMES. And if they don't suit your needs, you can fall back on the completely general DO. What more could you want?

Well, it turns out a handful of looping idioms come up over and over again, such as looping over various data structures: lists, vectors, hash tables, and packages. Or accumulating values in various ways while looping: collecting, counting, summing, minimizing, or maximizing. If you need a loop to do one of these things (or several at the same time), the LOOP macro may give you an easier way to express it.

The LOOP macro actually comes in two flavors--*simple* and *extended*. The simple version is as simple as can be--an infinite loop that doesn't bind any variables. The skeleton looks like this:

```
(loop
  body-form*)
```

The forms in body are evaluated each time through the loop, which will iterate forever unless you use RETURN to break out. For example, you could write the previous DO loop with a simple LOOP.

```
(loop
  (when (> (get-universal-time) *some-future-date*)
    (return))
  (format t "Waiting~%")
  (sleep 60))
```

The extended LOOP is quite a different beast. It's distinguished by the use of certain *loop keywords* that implement a special-purpose language for expressing looping idioms. It's worth noting that not all

Lispers love the extended LOOP language. At least one of Common Lisp's original designers hated it. LOOP's detractors complain that its syntax is totally un-Lispy (in other words, not enough parentheses). LOOP's fans counter that that's the point: complicated looping constructs are hard enough to understand without wrapping them up in DO's cryptic syntax. It's better, they say, to have a slightly more verbose syntax that gives you some clues what the heck is going on.

For instance, here's an idiomatic DO loop that collects the numbers from 1 to 10 into a list:

```
(do ((nums nil) (i 1 (1+ i)))
    ((> i 10) (nreverse nums))
  (push i nums)) ==> (1 2 3 4 5 6 7 8 9 10)
```

A seasoned Lisper won't have any trouble understanding that code--it's just a matter of understanding the basic form of a DO loop and recognizing the PUSH/NREVERSE idiom for building up a list. But it's not exactly transparent. The LOOP version, on the other hand, is almost understandable as an English sentence.

```
(loop for i from 1 to 10 collecting i) ==> (1 2 3 4 5 6 7 8 9 10)
```

The following are some more examples of simple uses of LOOP. This sums the first ten squares:

```
(loop for x from 1 to 10 summing (expt x 2)) ==> 385
```

This counts the number of vowels in a string:

```
(loop for x across "the quick brown fox jumps over the lazy dog"
      counting (find x "aeiou")) ==> 11
```

This computes the eleventh Fibonacci number, similar to the DO loop used earlier:

```
(loop for i below 10
      and a = 0 then b
      and b = 1 then (+ b a)
      finally (return  a))
```

The symbols `across`, `and`, `below`, `collecting`, `counting`, `finally`, `for`, `from`, `summing`, `then`, and `to` are some of the loop keywords whose presence identifies these as instances of the extended **LOOP**. [8]

I'll save the details of **LOOP** for Chapter 22, but it's worth noting here as another example of the way macros can be used to extend the base language. While **LOOP** provides its own language for expressing looping constructs, it doesn't cut you off from the rest of Lisp. The loop keywords are parsed according to loop's grammar, but the rest of the code in a **LOOP** is regular Lisp code.

And it's worth pointing out one more time that while the **LOOP** macro is quite a bit more complicated than macros such as **WHEN** or **UNLESS**, it *is* just another macro. If it hadn't been included in the standard library, you could implement it yourself or get a third-party library that does.

With that I'll conclude our tour of the basic control-construct macros. Now you're ready to take a closer look at how to define your own macros.

---

[1] To see what this misunderstanding looks like, find any longish Usenet thread cross-posted between comp.lang.lisp and any other comp.lang.* group with *macro* in the subject. A rough paraphrase goes like this:

Lispnik: "Lisp is the best because of its macros!";

Othernik: "You think Lisp is good *because of* macros?! But macros are horrible and evil; Lisp must be horrible and evil."

[2]Another important class of language constructs that are defined using macros are all the definitional constructs such as **DEFUN**, **DEFPARAMETER**, **DEFVAR**, and others. In Chapter 24 you'll define your own definitional macros that will allow you to concisely write code for reading and writing binary data.

[3]You can't actually feed this definition to Lisp because it's illegal to redefine names in the `COMMON-LISP` package where **WHEN** comes from. If you really want to try writing such a macro, you'd need to change the name to something else, such as `my-when`.

[4]The special operators, if you must know, are **TAGBODY** and **GO**. There's no need to discuss them now, but I'll cover them in Chapter 20.

[5]**DOLIST** is similar to Perl's `foreach` or Python's `for`. Java added a similar kind of loop construct with the "enhanced" `for` loop in Java 1.5, as part of JSR-201. Notice what a difference macros make. A Lisp programmer who notices a common pattern in their code can write a macro to give themselves a source-level abstraction of that pattern. A Java programmer who notices the same pattern has to convince Sun that this particular abstraction is worth adding to the language. Then Sun has to publish a JSR and convene an industry-wide "expert group" to hash everything out. That process--according to Sun--takes an average of 18 months. After that, the compiler writers all have to go upgrade their compilers to support the new feature. And even once the Java programmer's favorite compiler supports the new version of Java, they probably *still* can't use the new feature until they're allowed to break source compatibility with older versions of Java. So an annoyance that Common Lisp programmers can resolve for themselves within five minutes plagues Java programmers for years.

[6]A variant of **DO**, **DO***, assigns each variable its value before evaluating the step form for subsequent variables. For more details, consult your favorite Common Lisp reference.

[7]The **DOTIMES** is also preferred because the macro expansion will likely include declarations that allow the compiler to generate more efficient code.

[8]*Loop keywords* is a bit of a misnomer since they aren't keyword symbols. In fact, **LOOP** doesn't care what package the symbols are from. When the **LOOP** macro parses its body, it considers any appropriately named symbols equivalent. You could even use true keywords if you wanted--`:for`, `:across`, and so on--because they also have the correct name. But most folks just use plain symbols. Because the loop keywords are used only as syntactic markers, it doesn't matter if they're used for other purposes--as function or variable names.

Copyright Š 2003-2005, Peter Seibel

8. Macros: Defining Your Own

Now it's time to start writing your own macros. The standard macros I covered in the previous chapter hint at some of the things you can do with macros, but that's just the beginning. Common Lisp doesn't support macros so every Lisp programmer can create their own variants of standard control constructs any more than C supports functions so every C programmer can write trivial variants of the functions in the C standard library. Macros are part of the language to allow you to create abstractions on top of the core language and standard library that move you closer toward being able to directly express the things you want to express.

Perhaps the biggest barrier to a proper understanding of macros is, ironically, that they're so well integrated into the language. In many ways they seem like just a funny kind of function--they're written in Lisp, they take arguments and return results, and they allow you to abstract away distracting details. Yet despite these many similarities, macros operate at a different level than functions and create a totally different kind of abstraction.

Once you understand the difference between macros and functions, the tight integration of macros in the language will be a huge benefit. But in the meantime, it's a frequent source of confusion for new Lispers. The following story, while not true in a historical or technical sense, tries to alleviate the confusion by giving you a way to think about how macros work.

The Story of Mac: A Just-So Story

Once upon a time, long ago, there was a company of Lisp programmers. It was so long ago, in fact, that Lisp had no macros. Anything that couldn't be defined with a function or done with a special operator had to be written in full every time, which was rather a drag. Unfortunately, the programmers in this company--though brilliant--were also quite lazy. Often in the middle of their programs--when the tedium of writing a bunch of code got to be too much--they would instead write a note describing the code they needed to write at that place in the program. Even more unfortunately, because they were lazy, the programmers also hated to go back and actually write the code described by the notes. Soon the company had a big stack of programs that nobody could run because they were full of notes about code that still needed to be written.

In desperation, the big bosses hired a junior programmer, Mac, whose job was to find the notes, write the required code, and insert it into the program in place of the notes. Mac never ran the programs--they weren't done yet, of course, so he couldn't. But even if they had been completed, Mac wouldn't have known what inputs to feed them. So he just wrote his code based on the contents of the notes and sent it back to the original programmer.

With Mac's help, all the programs were soon completed, and the company made a ton of money selling them--so much money that the company could double the size of its programming staff. But for some reason no one thought to hire anyone to help Mac; soon he was single- handedly assisting several dozen programmers. To avoid

spending all his time searching for notes in source code, Mac made a small modification to the compiler the programmers used. Thereafter, whenever the compiler hit a note, it would e-mail him the note and wait for him to e-mail back the replacement code. Unfortunately, even with this change, Mac had a hard time keeping up with the programmers. He worked as carefully as he could, but sometimes-- especially when the notes weren't clear--he would make mistakes.

The programmers noticed, however, that the more precisely they wrote their notes, the more likely it was that Mac would send back correct code. One day, one of the programmers, having a hard time describing in words the code he wanted, included in one of his notes a Lisp program that would generate the code he wanted. That was fine by Mac; he just ran the program and sent the result to the compiler.

The next innovation came when a programmer put a note at the top of one of his programs containing a function definition and a comment that said, "Mac, don't write any code here, but keep this function for later; I'm going to use it in some of my other notes." Other notes in the same program said things such as, "Mac, replace this note with the result of running that other function with the symbols `x` and `y` as arguments."

This technique caught on so quickly that within a few days, most programs contained dozens of notes defining functions that were only used by code in other notes. To make it easy for Mac to pick out the notes containing only definitions that didn't require any

immediate response, the programmers tagged them with the standard preface: "Definition for Mac, Read Only." This--as the programmers were still quite lazy--was quickly shortened to "DEF. MAC. R/O" and then "DEFMACRO."

Pretty soon, there was no actual English left in the notes for Mac. All he did all day was read and respond to e-mails from the compiler containing DEFMACRO notes and calls to the functions defined in the DEFMACROs. Since the Lisp programs in the notes did all the real work, keeping up with the e-mails was no problem. Mac suddenly had a lot of time on his hands and would sit in his office daydreaming about white-sand beaches, clear blue ocean water, and drinks with little paper umbrellas in them.

Several months later the programmers realized nobody had seen Mac for quite some time. When they went to his office, they found a thin layer of dust over everything, a desk littered with travel brochures for various tropical locations, and the computer off. But the compiler still worked--how could it be? It turned out Mac had made one last change to the compiler: instead of e-mailing notes to Mac, the compiler now saved the functions defined by DEFMACRO notes and ran them when called for by the other notes. The programmers decided there was no reason to tell the big bosses Mac wasn't coming to the office anymore. So to this day, Mac draws a salary and from time to time sends the programmers a postcard from one tropical locale or another.

Macro Expansion Time vs. Runtime

The key to understanding macros is to be quite clear about the distinction between the code that generates code (macros) and the code that eventually makes up the program (everything else). When you write macros, you're writing programs that will be used by the compiler to generate the code that will then be compiled. Only after all the macros have been fully expanded and the resulting code compiled can the program actually be run. The time when macros run is called *macro expansion time*; this is distinct from *runtime*, when regular code, including the code generated by macros, runs.

It's important to keep this distinction firmly in mind because code running at macro expansion time runs in a very different environment than code running at runtime. Namely, at macro expansion time, there's no way to access the data that will exist at runtime. Like Mac, who couldn't run the programs he was working on because he didn't know what the correct inputs were, code running at macro expansion time can deal only with the data that's inherent in the source code. For instance, suppose the following source code appears somewhere in a program:

```
(defun foo (x)
  (when (> x 10) (print 'big)))
```

Normally you'd think of `x` as a variable that will hold the argument passed in a call to `foo`. But at macro expansion time, such as when the compiler is running the **`WHEN`** macro, the only data available is the source code. Since the program isn't running yet, there's no call to `foo` and thus no value associated with `x`. Instead, the values the

compiler passes to **WHEN** are the Lisp lists representing the source code, namely, `(> x 10)` and `(print 'big)`. Suppose that **WHEN** is defined, as you saw in the previous chapter, with something like the following macro:

```
(defmacro when (condition &rest body)
  `(if ,condition (progn ,@body)))
```

When the code in `foo` is compiled, the **WHEN** macro will be run with those two forms as arguments. The parameter `condition` will be bound to the form `(> x 10)`, and the form `(print 'big)` will be collected into a list that will become the value of the **&rest** `body` parameter. The backquote expression will then generate this code:

```
(if (> x 10) (progn (print 'big)))
```

by interpolating in the value of `condition` and splicing the value of `body` into the **PROGN**.

When Lisp is interpreted, rather than compiled, the distinction between macro expansion time and runtime is less clear because they're temporally intertwined. Also, the language standard doesn't specify exactly how an interpreter must handle macros--it could expand all the macros in the form being interpreted and then interpret the resulting code, or it could start right in on interpreting the form and expand macros when it hits them. In either case, macros are always passed the unevaluated Lisp objects representing the subforms of the macro form, and the job of the macro is still to produce code that will do something rather than to do anything directly.

DEFMACRO

As you saw in Chapter 3, macros really are defined with **DEFMACRO** forms, though it stands--of course--for DEFine MACRO, not Definition for Mac. The basic skeleton of a **DEFMACRO** is quite similar to the skeleton of a **DEFUN**.

```
(defmacro name (parameter*)
  "Optional documentation string."
  body-form*)
```

Like a function, a macro consists of a name, a parameter list, an optional documentation string, and a body of Lisp expressions.[1] However, as I just discussed, the job of a macro isn't to do anything directly--its job is to generate code that will later do what you want.

Macros can use the full power of Lisp to generate their expansion, which means in this chapter I can only scratch the surface of what you can do with macros. I can, however, describe a general process for writing macros that works for all macros from the simplest to the most complex.

The job of a macro is to translate a macro form--in other words, a Lisp form whose first element is the name of the macro--into code that does a particular thing. Sometimes you write a macro starting with the code you'd like to be able to write, that is, with an example macro form. Other times you decide to write a macro after you've written the same pattern of code several times and realize you can make your code clearer by abstracting the pattern.

Regardless of which end you start from, you need to figure out the other end before you can start writing a macro: you need to know both where you're coming from and where you're going before you can hope to write code to do it automatically. Thus, the first step of writing a macro is to write at least one example of a call to the macro and the code into which that call should expand.

Once you have an example call and the desired expansion, you're ready for the second step: writing the actual macro code. For simple macros this will be a trivial matter of writing a backquoted template with the macro parameters plugged into the right places. Complex macros will be significant programs in their own right, complete with helper functions and data structures.

After you've written code to translate the example call to the appropriate expansion, you need to make sure the abstraction the macro provides doesn't "leak" details of its implementation. Leaky macro abstractions will work fine for certain arguments but not others or will interact with code in the calling environment in undesirable ways. As it turns out, macros can leak in a small handful of ways, all of which are easily avoided as long as you know to check for them. I'll discuss how in the section "Plugging the Leaks."

To sum up, the steps to writing a macro are as follows:

1. Write a sample call to the macro and the code it should expand into, or vice versa.
2. Write code that generates the handwritten expansion from the arguments in the sample call.
3. Make sure the macro abstraction doesn't "leak."

A Sample Macro: do-primes

To see how this three-step process works, you'll write a macro `do-primes` that provides a looping construct similar to **DOTIMES** and **DOLIST** except that instead of iterating over integers or elements of a list, it iterates over successive prime numbers. This isn't meant to be an example of a particularly useful macro--it's just a vehicle for demonstrating the process.

First, you'll need two utility functions, one to test whether a given number is prime and another that returns the next prime number greater or equal to its argument. In both cases you can use a simple, but inefficient, brute-force approach.

```
(defun primep (number)
  (when (> number 1)
    (loop for fac from 2 to (isqrt number) never (zerop (mod number fac)))))

(defun next-prime (number)
  (loop for n from number when (primep n) return n))
```

Now you can write the macro. Following the procedure outlined previously, you need at least one example of a call to the macro and the code into which it should expand. Suppose you start with the idea that you want to be able to write this:

```
(do-primes (p 0 19)
  (format t "~d " p))
```

to express a loop that executes the body once each for each prime number greater or equal to 0 and less than or equal to 19, with the variable `p` holding the prime number. It makes sense to model this macro on the form of the standard **DOLIST** and **DOTIMES** macros; macros that follow the pattern of existing macros are easier to

understand and use than macros that introduce gratuitously novel syntax.

Without the `do-primes` macro, you could write such a loop with `DO` (and the two utility functions defined previously) like this:

```
(do ((p (next-prime 0) (next-prime (1+ p))))
    ((> p 19))
  (format t "~d " p))
```

Now you're ready to start writing the macro code that will translate from the former to the latter.

Macro Parameters

Since the arguments passed to a macro are Lisp objects representing the source code of the macro call, the first step in any macro is to extract whatever parts of those objects are needed to compute the expansion. For macros that simply interpolate their arguments directly into a template, this step is trivial: simply defining the right parameters to hold the different arguments is sufficient.

But this approach, it seems, will not suffice for `do-primes`. The first argument to the `do-primes` call is a list containing the name of the loop variable, `p`; the lower bound, `0`; and the upper bound, `19`. But if you look at the expansion, the list as a whole doesn't appear in the expansion; the three element are split up and put in different places.

You could define `do-primes` with two parameters, one to hold the list and a **`&rest`** parameter to hold the body forms, and then take apart the list by hand, something like this:

```
(defmacro do-primes (var-and-range &rest body)
  (let ((var (first var-and-range))
        (start (second var-and-range))
        (end (third var-and-range)))
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
         ((> ,var ,end))
       ,@body)))
```

In a moment I'll explain how the body generates the correct expansion; for now you can just note that the variables `var`, `start`, and `end` each hold a value, extracted from `var-and-range`, that's then interpolated into the backquote expression that generates `do-primes`'s expansion.

However, you don't need to take apart `var-and-range` "by hand" because macro parameter lists are what are called *destructuring* parameter lists. Destructuring, as the name suggests, involves taking apart a structure--in this case the list structure of the forms passed to a macro.

Within a destructuring parameter list, a simple parameter name can be replaced with a nested parameter list. The parameters in the nested parameter list will take their values from the elements of the expression that would have been bound to the parameter the list replaced. For instance, you can replace `var-and-range` with a list `(var start end)`, and the three elements of the list will automatically be destructured into those three parameters.

Another special feature of macro parameter lists is that you can use **`&body`** as a synonym for **`&rest`**. Semantically **`&body`** and **`&rest`** are equivalent, but many development environments will use the presence of a **`&body`** parameter to modify how they indent uses of the macro--typically **`&body`** parameters are used to hold a list of forms that make up the body of the macro.

So you can streamline the definition of `do-primes` and give a hint to both human readers and your development tools about its intended use by defining it like this:

```
(defmacro do-primes ((var start end) &body body)
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
       ((> ,var ,end))
     ,@body))
```

In addition to being more concise, destructuring parameter lists also give you automatic error checking--with `do-primes` defined this way,

Lisp will be able to detect a call whose first argument isn't a three-element list and will give you a meaningful error message just as if you had called a function with too few or too many arguments. Also, in development environments such as SLIME that indicate what arguments are expected as soon as you type the name of a function or macro, if you use a destructuring parameter list, the environment will be able to tell you more specifically the syntax of the macro call. With the original definition, SLIME would tell you `do-primes` is called like this:

```
(do-primes var-and-range &rest body)
```

But with the new definition, it can tell you that a call should look like this:

```
(do-primes (var start end) &body body)
```

Destructuring parameter lists can contain **`&optional`**, **`&key`**, and **`&rest`** parameters and can contain nested destructuring lists. However, you don't need any of those options to write `do-primes`.

Generating the Expansion

Because `do-primes` is a fairly simple macro, after you've destructured the arguments, all that's left is to interpolate them into a template to get the expansion.

For simple macros like `do-primes`, the special backquote syntax is perfect. To review, a backquoted expression is similar to a quoted expression except you can "unquote" particular subexpressions by preceding them with a comma, possibly followed by an at (@) sign. Without an at sign, the comma causes the value of the subexpression to be included as is. With an at sign, the value--which must be a list--is "spliced" into the enclosing list.

Another useful way to think about the backquote syntax is as a particularly concise way of writing code that generates lists. This way of thinking about it has the benefit of being pretty much exactly what's happening under the covers--when the reader reads a backquoted expression, it translates it into code that generates the appropriate list structure. For instance, `` `(,a b) `` might be read as `(list a 'b)`. The language standard doesn't specify exactly what code the reader must produce as long as it generates the right list structure.

Table 8-1 shows some examples of backquoted expressions along with equivalent list-building code and the result you'd get if you evaluated either the backquoted expression or the equivalent code.[2]

Table 8-1. Backquote Examples

Backquote Syntax	Equivalent List-Building Code	Result
`(a (+ 1 2) c)	(list 'a '(+ 1 2) 'c)	(a (+ 1 2) c)
`(a ,(+ 1 2) c)	(list 'a (+ 1 2) 'c)	(a 3 c)
`(a (list 1 2) c)	(list 'a '(list 1 2) 'c)	(a (list 1 2) c)
`(a ,(list 1 2) c)	(list 'a (list 1 2) 'c)	(a (1 2) c)
`(a ,@(list 1 2) c)	(append (list 'a) (list 1 2) (list 'c))	(a 1 2 c)

It's important to note that backquote is just a convenience. But it's a big convenience. To appreciate how big, compare the backquoted version of `do-primes` to the following version, which uses explicit list-building code:

```
(defmacro do-primes-a ((var start end) &body body)
  (append '(do)
          (list  (list (list var
                             (list 'next-prime start)
                             (list 'next-prime (list '1+ var)))))
          (list (list (list '> var end)))
          body))
```

As you'll see in a moment, the current implementation of `do-primes` doesn't handle certain edge cases correctly. But first you should verify that it at least works for the original example. You can test it in two ways. You can test it indirectly by simply using it--presumably, if the resulting behavior is correct, the expansion is correct. For instance, you can type the original example's use of `do-primes` to the REPL and see that it indeed prints the right series of prime numbers.

```
CL-USER> (do-primes (p 0 19) (format t "~d " p))
2 3 5 7 11 13 17 19
NIL
```

Or you can check the macro directly by looking at the expansion of a particular call. The function **`MACROEXPAND-1`** takes any Lisp expression as an argument and returns the result of doing one level of macro expansion.[3] Because **`MACROEXPAND-1`** is a function, to pass it a literal

macro form you must quote it. You can use it to see the expansion of the previous call.[4]

```
CL-USER> (macroexpand-1 '(do-primes (p 0 19) (format t "~d " p)))
(DO ((P (NEXT-PRIME 0) (NEXT-PRIME (1+ P))))
    ((> P 19))
  (FORMAT T "~d " P))
T
```

Or, more conveniently, in SLIME you can check a macro's expansion by placing the cursor on the opening parenthesis of a macro form in your source code and typing `C-c RET` to invoke the Emacs function `slime-macroexpand-1`, which will pass the macro form to **`MACROEXPAND-1`** and "pretty print" the result in a temporary buffer.

However you get to it, you can see that the result of macro expansion is the same as the original handwritten expansion, so it seems that `do-primes` works.

Plugging the Leaks

In his essay "The Law of Leaky Abstractions," Joel Spolsky coined the term *leaky abstraction* to describe an abstraction that "leaks" details it's supposed to be abstracting away. Since writing a macro is a way of creating an abstraction, you need to make sure your macros don't leak needlessly.[5]

As it turns out, a macro can leak details of its inner workings in three ways. Luckily, it's pretty easy to tell whether a given macro suffers from any of those leaks and to fix them.

The current definition suffers from one of the three possible macro leaks: namely, it evaluates the `end` subform too many times. Suppose you were to call `do-primes` with, instead of a literal number such as `19`, an expression such as `(random 100)` in the `end` position.

```
(do-primes (p 0 (random 100))
  (format t "~d " p))
```

Presumably the intent here is to loop over the primes from zero to whatever random number is returned by `(random 100)`. However, this isn't what the current implementation does, as **`MACROEXPAND-1`** shows.

```
CL-USER> (macroexpand-1 '(do-primes (p 0 (random 100)) (format t "~d " p)))
(DO ((P (NEXT-PRIME 0) (NEXT-PRIME (1+ P))))
    ((> P (RANDOM 100)))
  (FORMAT T "~d " P))
T
```

When this expansion code is run, **`RANDOM`** will be called each time the end test for the loop is evaluated. Thus, instead of looping until `p` is greater than an initially chosen random number, this loop will iterate until it happens to draw a random number less than or equal to the

current value of `p`. While the total number of iterations will still be random, it will be drawn from a much different distribution than the uniform distribution **RANDOM** returns.

This is a leak in the abstraction because, to use the macro correctly, the caller needs to be aware that the `end` form is going to be evaluated more than once. One way to plug this leak would be to simply define this as the behavior of `do-primes`. But that's not very satisfactory--you should try to observe the Principle of Least Astonishment when implementing macros. And programmers will typically expect the forms they pass to macros to be evaluated no more times than absolutely necessary.[6] Furthermore, since `do-primes` is built on the model of the standard macros, **DOTIMES** and **DOLIST**, neither of which causes any of the forms except those in the body to be evaluated more than once, most programmers will expect `do-primes` to behave similarly.

You can fix the multiple evaluation easily enough; you just need to generate code that evaluates `end` once and saves the value in a variable to be used later. Recall that in a **DO** loop, variables defined with an initialization form and no step form don't change from iteration to iteration. So you can fix the multiple evaluation problem with this definition:

```
(defmacro do-primes ((var start end) &body body)
  `(do ((ending-value ,end)
        (,var (next-prime ,start) (next-prime (1+ ,var))))
       ((> ,var ending-value))
     ,@body))
```

Unfortunately, this fix introduces two new leaks to the macro abstraction.

One new leak is similar to the multiple-evaluation leak you just fixed. Because the initialization forms for variables in a `DO` loop are evaluated in the order the variables are defined, when the macro expansion is evaluated, the expression passed as `end` will be evaluated before the expression passed as `start`, opposite to the order they appear in the macro call. This leak doesn't cause any problems when `start` and `end` are literal values like 0 and 19. But when they're forms that can have side effects, evaluating them out of order can once again run afoul of the Principle of Least Astonishment.

This leak is trivially plugged by swapping the order of the two variable definitions.

```
(defmacro do-primes ((var start end) &body body)
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
        (ending-value ,end))
       ((> ,var ending-value))
     ,@body))
```

The last leak you need to plug was created by using the variable name `ending-value`. The problem is that the name, which ought to be a purely internal detail of the macro implementation, can end up interacting with code passed to the macro or in the context where the macro is called. The following seemingly innocent call to `do-primes` doesn't work correctly because of this leak:

```
(do-primes (ending-value 0 10)
  (print ending-value))
```

Neither does this one:

```
(let ((ending-value 0))
  (do-primes (p 0 10)
    (incf ending-value p))
  ending-value)
```

Again, **MACROEXPAND-1** can show you the problem. The first call expands to this:

```
(do ((ending-value (next-prime 0) (next-prime (1+ ending-value)))
     (ending-value 10))
    ((> ending-value ending-value))
  (print ending-value))
```

Some Lisps may reject this code because `ending-value` is used twice as a variable name in the same **DO** loop. If not rejected outright, the code will loop forever since `ending-value` will never be greater than itself.

The second problem call expands to the following:

```
(let ((ending-value 0))
  (do ((p (next-prime 0) (next-prime (1+ p)))
       (ending-value 10))
      ((> p ending-value))
    (incf ending-value p))
  ending-value)
```

In this case the generated code is perfectly legal, but the behavior isn't at all what you want. Because the binding of `ending-value` established by the **LET** outside the loop is shadowed by the variable with the same name inside the **DO**, the form `(incf ending-value p)` increments the loop variable `ending-value` instead of the outer variable with the same name, creating another infinite loop.[7]

Clearly, what you need to patch this leak is a symbol that will never be used outside the code generated by the macro. You could try using a really unlikely name, but that's no guarantee. You could also protect yourself to some extent by using packages, as described in Chapter 21. But there's a better solution.

The function **GENSYM** returns a unique symbol each time it's called. This is a symbol that has never been read by the Lisp reader and never will be because it isn't interned in any package. Thus, instead of using a literal name like `ending-value`, you can generate a new symbol each time `do-primes` is expanded.

```
(defmacro do-primes ((var start end) &body body)
  (let ((ending-value-name (gensym)))
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
          (,ending-value-name ,end))
         ((> ,var ,ending-value-name))
       ,@body)))
```

Note that the code that calls **GENSYM** isn't part of the expansion; it runs as part of the macro expander and thus creates a new symbol each time the macro is expanded. This may seem a bit strange at first--`ending-value-name` is a variable whose value is the name of another variable. But really it's no different from the parameter `var` whose value is the name of a variable--the difference is the value of `var` was created by the reader when the macro form was read, and the value of `ending-value-name` is generated programmatically when the macro code runs.

With this definition the two previously problematic forms expand into code that works the way you want. The first form:

```
(do-primes (ending-value 0 10)
  (print ending-value))
```

expands into the following:

```
(do ((ending-value (next-prime 0) (next-prime (1+ ending-value)))
     (#:g2141 10))
    ((> ending-value #:g2141))
  (print ending-value))
```

Now the variable used to hold the ending value is the gensymed symbol, `#:g2141`. The name of the symbol, `G2141`, was generated by **`GENSYM`** but isn't significant; the thing that matters is the object identity of the symbol. Gensymed symbols are printed in the normal syntax for uninterned symbols, with a leading `#:`.

The other previously problematic form:

```
(let ((ending-value 0))
  (do-primes (p 0 10)
    (incf ending-value p))
  ending-value)
```

looks like this if you replace the `do-primes` form with its expansion:

```
(let ((ending-value 0))
  (do ((p (next-prime 0) (next-prime (1+ p)))
       (#:g2140 10))
      ((> p #:g2140))
    (incf ending-value p))
  ending-value)
```

Again, there's no leak since the `ending-value` variable bound by the **`LET`** surrounding the `do-primes` loop is no longer shadowed by any variables introduced in the expanded code.

Not all literal names used in a macro expansion will necessarily cause a problem--as you get more experience with the various binding forms, you'll be able to determine whether a given name is being used in a position that could cause a leak in a macro abstraction. But there's no real downside to using a gensymed name just to be safe.

With that fix, you've plugged all the leaks in the implementation of `do-primes`. Once you've gotten a bit of macro-writing experience under your belt, you'll learn to write macros with these kinds of leaks

preplugged. It's actually fairly simple if you follow these rules of thumb:

- Unless there's a particular reason to do otherwise, include any subforms in the expansion in positions that will be evaluated in the same order as the subforms appear in the macro call.
- Unless there's a particular reason to do otherwise, make sure subforms are evaluated only once by creating a variable in the expansion to hold the value of evaluating the argument form and then using that variable anywhere else the value is needed in the expansion.
- Use `GENSYM` at macro expansion time to create variable names used in the expansion.

Macro-Writing Macros

Of course, there's no reason you should be able to take advantage of macros only when writing functions. The job of macros is to abstract away common syntactic patterns, and certain patterns come up again and again in writing macros that can also benefit from being abstracted away.

In fact, you've already seen one such pattern--many macros will, like the last version of `do-primes`, start with a **`LET`** that introduces a few variables holding gensymed symbols to be used in the macro's expansion. Since this is such a common pattern, why not abstract it away with its own macro?

In this section you'll write a macro, `with-gensyms`, that does just that. In other words, you'll write a macro-writing macro: a macro that generates code that generates code. While complex macro-writing macros can be a bit confusing until you get used to keeping the various levels of code clear in your mind, `with-gensyms` is fairly straightforward and will serve as a useful but not too strenuous mental limbering exercise.

You want to be able to write something like this:

```
(defmacro do-primes ((var start end) &body body)
  (with-gensyms (ending-value-name)
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
          (,ending-value-name ,end))
         ((> ,var ,ending-value-name))
       ,@body)))
```

and have it be equivalent to the previous version of `do-primes`. In other words, the `with-gensyms` needs to expand into a **`LET`** that binds

each named variable, `ending-value-name` in this case, to a gensymed symbol. That's easy enough to write with a simple backquote template.

```
(defmacro with-gensyms ((&rest names) &body body)
  `(let ,(loop for n in names collect `(,n (gensym)))
     ,@body))
```

Note how you can use a comma to interpolate the value of the **LOOP** expression. The loop generates a list of binding forms where each binding form consists of a list containing one of the names given to `with-gensyms` and the literal code `(gensym)`. You can test what code the **LOOP** expression would generate at the REPL by replacing `names` with a list of symbols.

```
CL-USER> (loop for n in '(a b c) collect `(,n (gensym)))
((A (GENSYM)) (B (GENSYM)) (C (GENSYM)))
```

After the list of binding forms, the body argument to `with-gensyms` is spliced in as the body of the **LET**. Thus, in the code you wrap in a `with-gensyms` you can refer to any of the variables named in the list of variables passed to `with-gensyms`.

If you macro-expand the `with-gensyms` form in the new definition of `do-primes`, you should see something like this:

```
(let ((ending-value-name (gensym)))
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
        (,ending-value-name ,end))
       ((> ,var ,ending-value-name))
     ,@body))
```

Looks good. While this macro is fairly trivial, it's important to keep clear about when the different macros are expanded: when you compile the **DEFMACRO** of `do-primes`, the `with-gensyms` form is expanded into the code just shown and compiled. Thus, the compiled version of

`do-primes` is just the same as if you had written the outer **`LET`** by hand. When you compile a function that uses `do-primes`, the code *generated* by `with-gensyms` runs generating the `do-primes` expansion, but `with-gensyms` itself isn't needed to compile a `do-primes` form since it has already been expanded, back when `do-primes` was compiled.

Another classic macro-writing MACRO: ONCE-ONLY

Another classic macro-writing macro is `once-only`, which is used to generate code that evaluates certain macro arguments once only and in a particular order. Using `once-only`, you could write `do-primes` almost as simply as the original leaky version, like this:

```
(defmacro do-primes ((var start end) &body body)
  (once-only (start end)
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
         ((> ,var ,end))
       ,@body)))
```

However, the implementation of `once-only` is a bit too involved for a blow-by-blow explanation, as it relies on multiple levels of backquoting and unquoting. If you really want to sharpen your macro chops, you can try to figure out how it works. It looks like this:

```
(defmacro once-only ((&rest names) &body body)
  (let ((gensyms (loop for n in names collect (gensym))))
    `(let (,@(loop for g in gensyms collect `(,g (gensym))))
       `(let (,,@(loop for g in gensyms for n in names collect ``(,,g ,,n)))
          ,(let (,@(loop for n in names for g in gensyms collect `(,n ,g)))
             ,@body)))))
```

Beyond Simple Macros

I could, of course, say a lot more about macros. All the macros you've seen so far have been fairly simple examples that save you a bit of typing but don't provide radical new ways of expressing things. In upcoming chapters you'll see examples of macros that allow you to express things in ways that would be virtually impossible without macros. You'll start in the very next chapter, in which you'll build a simple but effective unit test framework.

---

[1]As with functions, macros can also contain declarations, but you don't need to worry about those for now.

[2]**APPEND**, which I haven't discussed yet, is a function that takes any number of list arguments and returns the result of splicing them together into a single list.

[3]Another function, **MACROEXPAND**, keeps expanding the result as long as the first element of the resulting expansion is the name of the macro. However, this will often show you a much lower-level view of what the code is doing than you want, since basic control constructs such as DO are also implemented as macros. In other words, while it can be educational to see what your macro ultimately expands into, it isn't a very useful view into what your own macros are doing.

[4]If the macro expansion is shown all on one line, it's probably because the variable ***PRINT-PRETTY*** is **NIL**. If it is, evaluating `(setf *print-pretty* t)` should make the macro expansion easier to read.

[5]This is from *Joel on Software* by Joel Spolsky, also available at `http://www.joelonsoftware.com/ articles/LeakyAbstractions.html`. Spolsky's point in the essay is that all abstractions leak to some extent; that is, there are no perfect abstractions. But that doesn't mean you should tolerate leaks you can easily plug.

[6]Of course, certain forms are supposed to be evaluated more than once, such as the forms in the body of a `do-primes` loop.

[7]It may not be obvious that this loop is necessarily infinite given the nonuniform occurrences of prime numbers. The starting point for a proof that it is in fact infinite is Bertrand's postulate, which says for any $n > 1$, there exists a prime p, $n < p < 2n$. From there you can prove that for any prime number, P less than the sum of the preceding prime numbers, the next prime, P', is also smaller than the original sum plus P.

Copyright © 2003-2005, Peter Seibel

9. Practical: Building a Unit Test Framework

In this chapter you'll return to cutting code and develop a simple unit testing framework for Lisp. This will give you a chance to use some of the features you've learned about since Chapter 3, including macros and dynamic variables, in real code.

The main design goal of the test framework will be to make it as easy as possible to add new tests, to run various suites of tests, and to track down test failures. For now you'll focus on designing a framework you can use during interactive development.

The key feature of an automated testing framework is that the framework is responsible for telling you whether all the tests passed. You don't want to spend your time slogging through test output checking answers when the computer can do it much more quickly and accurately. Consequently, each test case must be an expression that yields a boolean value--true or false, pass or fail. For instance, if you were writing tests for the built-in + function, these might be reasonable test cases:[1]

```
(= (+ 1 2) 3)
(= (+ 1 2 3) 6)
(= (+ -1 -3) -4)
```

Functions that have side effects will be tested slightly differently--you'll have to call the function and then check for evidence of the

expected side effects.[2] But in the end, every test case has to boil down to a boolean expression, thumbs up or thumbs down.

Two First Tries

If you were doing ad hoc testing, you could enter these expressions at the REPL and check that they return **T**. But you want a framework that makes it easy to organize and run these test cases whenever you want. If you want to start with the simplest thing that could possibly work, you can just write a function that evaluates the test cases and **AND**s the results together.

```
(defun test-+ ()
  (and
    (= (+ 1 2) 3)
    (= (+ 1 2 3) 6)
    (= (+ -1 -3) -4)))
```

Whenever you want to run this set of test cases, you can call `test-+`.

```
CL-USER> (test-+)
T
```

As long as it returns **T**, you know the test cases are passing. This way of organizing tests is also pleasantly concise--you don't have to write a bunch of test bookkeeping code. However, as you'll discover the first time a test case fails, the result reporting leaves something to be desired. When `test-+` returns **NIL**, you'll know something failed, but you'll have no idea which test case it was.

So let's try another simple--even simpleminded--approach. To find out what happens to each test case, you could write something like this:

```
(defun test-+ ()
  (format t "~:[FAIL~;pass~] ... ~a~%" (= (+ 1 2) 3) '(= (+ 1 2) 3))
  (format t "~:[FAIL~;pass~] ... ~a~%" (= (+ 1 2 3) 6) '(= (+ 1 2 3) 6))
  (format t "~:[FAIL~;pass~] ... ~a~%" (= (+ -1 -3) -4) '(= (+ -1 -3) -4)))
```

Now each test case will be reported individually. The `~:[FAIL~;pass~]` part of the **FORMAT** directive causes **FORMAT** to print "FAIL" if the first format argument is false and "pass" otherwise.[3] Then you label the result with the test expression itself. Now running `test-+` shows you exactly what's going on.

```
CL-USER> (test-+)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
pass ... (= (+ -1 -3) -4)
NIL
```

This time the result reporting is more like what you want, but the code itself is pretty gross. The repeated calls to **FORMAT** as well as the tedious duplication of the test expression cry out to be refactored. The duplication of the test expression is particularly grating because if you mistype it, the test results will be mislabeled.

Another problem is that you don't get a single indicator whether all the test cases passed. It's easy enough, with only three test cases, to scan the output looking for "FAIL"; however, when you have hundreds of test cases, it'll be more of a hassle.

Refactoring

What you'd really like is a way to write test functions as streamlined as the first `test-+` that return a single **`T`** or **`NIL`** value but that also report on the results of individual test cases like the second version. Since the second version is close to what you want in terms of functionality, your best bet is to see if you can factor out some of the annoying duplication.

The simplest way to get rid of the repeated similar calls to **`FORMAT`** is to create a new function.

```
(defun report-result (result form)
  (format t "~:[FAIL~;pass~] ... ~a~%" result form))
```

Now you can write `test-+` with calls to `report-result` instead of **`FORMAT`**. It's not a huge improvement, but at least now if you decide to change the way you report results, there's only one place you have to change.

```
(defun test-+ ()
  (report-result (= (+ 1 2) 3) '(= (+ 1 2) 3))
  (report-result (= (+ 1 2 3) 6) '(= (+ 1 2 3) 6))
  (report-result (= (+ -1 -3) -4) '(= (+ -1 -3) -4)))
```

Next you need to get rid of the duplication of the test case expression, with its attendant risk of mislabeling of results. What you'd really like is to be able to treat the expression as both code (to get the result) and data (to use as the label). Whenever you want to treat code as data, that's a sure sign you need a macro. Or, to look at it another way, what you need is a way to automate writing the error-prone `report-result` calls. You'd like to be able to say something like this:

```
(check (= (+ 1 2) 3))
```

and have it mean the following:

```
(report-result (= (+ 1 2) 3) '(= (+ 1 2) 3))
```

Writing a macro to do this translation is trivial.

```
(defmacro check (form)
  `(report-result ,form ',form))
```

Now you can change `test-+` to use `check`.

```
(defun test-+ ()
  (check (= (+ 1 2) 3))
  (check (= (+ 1 2 3) 6))
  (check (= (+ -1 -3) -4)))
```

Since you're on the hunt for duplication, why not get rid of those repeated calls to `check`? You can define `check` to take an arbitrary number of forms and wrap them each in a call to `report-result`.

```
(defmacro check (&body forms)
  `(progn
     ,@(loop for f in forms collect `(report-result ,f ',f))))
```

This definition uses a common macro idiom of wrapping a **PROGN** around a series of forms in order to turn them into a single form. Notice also how you can use `,@` to splice in the result of an expression that returns a list of expressions that are themselves generated with a backquote template.

With the new version of `check` you can write a new version of `test-+` like this:

```
(defun test-+ ()
  (check
    (= (+ 1 2) 3)
    (= (+ 1 2 3) 6)
    (= (+ -1 -3) -4)))
```

that is equivalent to the following code:

```
(defun test-+ ()
  (progn
    (report-result (= (+ 1 2) 3) '(= (+ 1 2) 3))
    (report-result (= (+ 1 2 3) 6) '(= (+ 1 2 3) 6))
    (report-result (= (+ -1 -3) -4) '(= (+ -1 -3) -4))))
```

Thanks to `check`, this version is as concise as the first version of `test-+` but expands into code that does the same thing as the second version. And now any changes you want to make to how `test-+` behaves, you can make by changing `check`.

Fixing the Return Value

You can start with fixing `test-+` so its return value indicates whether all the test cases passed. Since `check` is responsible for generating the code that ultimately runs the test cases, you just need to change it to generate code that also keeps track of the results.

As a first step, you can make a small change to `report-result` so it returns the result of the test case it's reporting.

```
(defun report-result (result form)
  (format t "~:[FAIL~;pass~] ... ~a~%" result form)
  result)
```

Now that `report-result` returns the result of its test case, it might seem you could just change the **PROGN** to an **AND** to combine the results. Unfortunately, **AND** doesn't do quite what you want in this case because of its short-circuiting behavior: as soon as one test case fails, **AND** will skip the rest. On the other hand, if you had a construct that worked like **AND** without the short-circuiting, you could use it in the place of **PROGN**, and you'd be done. Common Lisp doesn't provide such a construct, but that's no reason you can't use it: it's a trivial matter to write a macro to provide it yourself.

Leaving test cases aside for a moment, what you want is a macro--let's call it `combine-results`--that will let you say this:

```
(combine-results
  (foo)
  (bar)
  (baz))
```

and have it mean something like this:

```
(let ((result t))
  (unless (foo) (setf result nil))
  (unless (bar) (setf result nil))
  (unless (baz) (setf result nil))
  result)
```

The only tricky bit to writing this macro is that you need to introduce a variable--`result` in the previous code--in the expansion. As you saw in the previous chapter, using a literal name for variables in macro expansions can introduce a leak in your macro abstraction, so you'll need to create a unique name. This is a job for `with-gensyms`. You can define `combine-results` like this:

```
(defmacro combine-results (&body forms)
  (with-gensyms (result)
    `(let ((,result t))
      ,@(loop for f in forms collect `(unless ,f (setf ,result nil)))
      ,result)))
```

Now you can fix `check` by simply changing the expansion to use `combine-results` instead of **`PROGN`**.

```
(defmacro check (&body forms)
  `(combine-results
    ,@(loop for f in forms collect `(report-result ,f ',f))))
```

With that version of `check`, `test-+` should emit the results of its three test expressions and then return **`T`** to indicate that everything passed.[4]

```
CL-USER> (test-+)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
pass ... (= (+ -1 -3) -4)
T
```

And if you change one of the test cases so it fails,[5] the final return value changes to **`NIL`**.

```
CL-USER> (test-+)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
```

```
FAIL ... (= (+ -1 -3) -5)
NIL
```

Better Result Reporting

As long as you have only one test function, the current result reporting is pretty clear. If a particular test case fails, all you have to do is find the test case in the `check` form and figure out why it's failing. But if you write a lot of tests, you'll probably want to organize them somehow, rather than shoving them all into one function. For instance, suppose you wanted to add some test cases for the `*` function. You might write a new test function.

```
(defun test-* ()
  (check
    (= (* 2 2) 4)
    (= (* 3 5) 15)))
```

Now that you have two test functions, you'll probably want another function that runs all the tests. That's easy enough.

```
(defun test-arithmetic ()
  (combine-results
   (test-+)
   (test-*)))
```

In this function you use `combine-results` instead of `check` since both `test-+` and `test-*` will take care of reporting their own results. When you run `test-arithmetic`, you'll get the following results:

```
CL-USER> (test-arithmetic)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
pass ... (= (+ -1 -3) -4)
pass ... (= (* 2 2) 4)
pass ... (= (* 3 5) 15)
T
```

Now imagine that one of the test cases failed and you need to track down the problem. With only five test cases and two test functions, it won't be too hard to find the code of the failing test case. But

suppose you had 500 test cases spread across 20 functions. It might be nice if the results told you what function each test case came from.

Since the code that prints the results is centralized in `report-result`, you need a way to pass information about what test function you're in to `report-result`. You could add a parameter to `report-result` to pass this information, but `check`, which generates the calls to `report-result`, doesn't know what function it's being called from, which means you'd also have to change the way you call `check`, passing it an argument that it simply passes onto `report-result`.

This is exactly the kind of problem dynamic variables were designed to solve. If you create a dynamic variable that each test function binds to the name of the function before calling `check`, then `report-result` can use it without `check` having to know anything about it.

Step one is to declare the variable at the top level.

```
(defvar *test-name* nil)
```

Now you need to make another tiny change to `report-result` to include `*test-name*` in the **`FORMAT`** output.

```
(format t "~:[FAIL~;pass~] ... ~a: ~a~%" result *test-name* form)
```

With those changes, the test functions will still work but will produce the following output because `*test-name*` is never rebound:

```
CL-USER> (test-arithmetic)
pass ... NIL: (= (+ 1 2) 3)
pass ... NIL: (= (+ 1 2 3) 6)
pass ... NIL: (= (+ -1 -3) -4)
pass ... NIL: (= (* 2 2) 4)
```

```
pass ... NIL: (= (* 3 5) 15)
T
```

For the name to be reported properly, you need to change the two test functions.

```
(defun test-+ ()
  (let ((*test-name* 'test-+))
    (check
      (= (+ 1 2) 3)
      (= (+ 1 2 3) 6)
      (= (+ -1 -3) -4))))

(defun test-* ()
  (let ((*test-name* 'test-*))
    (check
      (= (* 2 2) 4)
      (= (* 3 5) 15))))
```

Now the results are properly labeled.

```
CL-USER> (test-arithmetic)
pass ... TEST-+: (= (+ 1 2) 3)
pass ... TEST-+: (= (+ 1 2 3) 6)
pass ... TEST-+: (= (+ -1 -3) -4)
pass ... TEST-*: (= (* 2 2) 4)
pass ... TEST-*: (= (* 3 5) 15)
T
```

An Abstraction Emerges

In fixing the test functions, you've introduced several new bits of duplication. Not only does each function have to include the name of the function twice--once as the name in the `DEFUN` and once in the binding of `*test-name*`--but the same three-line code pattern is duplicated between the two functions. You could remove the duplication simply on the grounds that duplication is bad. But if you look more closely at the root cause of the duplication, you can learn an important lesson about how to use macros.

The reason both these functions start the same way is because they're both test functions. The duplication arises because, at the moment, *test function* is only half an abstraction. The abstraction exists in your mind, but in the code there's no way to express "this is a test function" other than to write code that follows a particular pattern.

Unfortunately, partial abstractions are a crummy tool for building software. Because a half abstraction is expressed in code by a manifestation of the pattern, you're guaranteed to have massive code duplication with all the normal bad consequences that implies for maintainability. More subtly, because the abstraction exists only in the minds of programmers, there's no mechanism to make sure different programmers (or even the same programmer working at different times) actually understand the abstraction the same way. To make a complete abstraction, you need a way to express "this is a test function" and have all the code required by the pattern be generated for you. In other words, you need a macro.

Because the pattern you're trying to capture is a **DEFUN** plus some boilerplate code, you need to write a macro that will expand into a **DEFUN**. You'll then use this macro, instead of a plain **DEFUN** to define test functions, so it makes sense to call it `deftest`.

```
(defmacro deftest (name parameters &body body)
  `(defun ,name ,parameters
    (let ((*test-name* ',name))
      ,@body)))
```

With this macro you can rewrite `test-+` as follows:

```
(deftest test-+ ()
  (check
    (= (+ 1 2) 3)
    (= (+ 1 2 3) 6)
    (= (+ -1 -3) -4)))
```

A Hierarchy of Tests

Now that you've established test functions as first-class citizens, the question might arise, should `test-arithmetic` be a test function? As things stand, it doesn't really matter--if you did define it with `deftest`, its binding of `*test-name*` would be shadowed by the bindings in `test-+` and `test-*` before any results are reported.

But now imagine you've got thousands of test cases to organize. The first level of organization is provided by test functions such as `test-+` and `test-*` that directly call `check`. But with thousands of test cases, you'll likely need other levels of organization. Functions such as `test-arithmetic` can group related test functions into test suites. Now suppose some low-level test functions are called from multiple test suites. It's not unheard of for a test case to pass in one context but fail in another. If that happens, you'll probably want to know more than just what low-level test function contains the test case.

If you define the test suite functions such as `test-arithmetic` with `deftest` and make a small change to the `*test-name*` bookkeeping, you can have results reported with a "fully qualified" path to the test case, something like this:

```
pass ... (TEST-ARITHMETIC TEST-+): (= (+ 1 2) 3)
```

Because you've already abstracted the process of defining a test function, you can change the bookkeeping details without modifying the code of the test functions.[6] To make `*test-name*` hold a list of test function names instead of just the name of the most recently entered test function, you just need to change this binding form:

```
(let ((*test-name* ',name))
```

to the following:

```
(let ((*test-name* (append *test-name* (list ',name))))
```

Since **APPEND** returns a new list made up of the elements of its arguments, this version will bind `*test-name*` to a list containing the old contents of `*test-name*` with the new name tacked onto the end.[7] When each test function returns, the old value of `*test-name*` will be restored.

Now you can redefine `test-arithmetic` with `deftest` instead of **DEFUN**.

```
(deftest test-arithmetic ()
  (combine-results
   (test-+)
   (test-*)))
```

The results now show exactly how you got to each test expression.

```
CL-USER> (test-arithmetic)
pass ... (TEST-ARITHMETIC TEST-+): (= (+ 1 2) 3)
pass ... (TEST-ARITHMETIC TEST-+): (= (+ 1 2 3) 6)
pass ... (TEST-ARITHMETIC TEST-+): (= (+ -1 -3) -4)
pass ... (TEST-ARITHMETIC TEST-*): (= (* 2 2) 4)
pass ... (TEST-ARITHMETIC TEST-*): (= (* 3 5) 15)
T
```

As your test suite grows, you can add new layers of test functions; as long as they're defined with `deftest`, the results will be reported correctly. For instance, the following:

```
(deftest test-math ()
  (test-arithmetic))
```

would generate these results:

```
CL-USER> (test-math)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-+): (= (+ 1 2) 3)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-+): (= (+ 1 2 3) 6)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-+): (= (+ -1 -3) -4)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-*): (= (* 2 2) 4)
```

```
pass ... (TEST-MATH TEST-ARITHMETIC TEST-*): (= (* 3 5) 15)
T
```

Wrapping Up

You could keep going, adding more features to this test framework. But as a framework for writing tests with a minimum of busywork and easily running them from the REPL, this is a reasonable start. Here's the complete code, all 26 lines of it:

```
(defvar *test-name* nil)

(defmacro deftest (name parameters &body body)
  "Define a test function. Within a test function we can call
   other test functions or use 'check' to run individual test
   cases."
  `(defun ,name ,parameters
     (let ((*test-name* (append *test-name* (list ',name))))
       ,@body)))

(defmacro check (&body forms)
  "Run each expression in 'forms' as a test case."
  `(combine-results
     ,@(loop for f in forms collect `(report-result ,f ',f))))

(defmacro combine-results (&body forms)
  "Combine the results (as booleans) of evaluating 'forms' in order."
  (with-gensyms (result)
    `(let ((,result t))
       ,@(loop for f in forms collect `(unless ,f (setf ,result nil)))
       ,result)))

(defun report-result (result form)
  "Report the results of a single test case. Called by 'check'."
  (format t "~:[FAIL~;pass~] ... ~a: ~a~%" result *test-name* form)
  result)
```

It's worth reviewing how you got here because it's illustrative of how programming in Lisp often goes.

You started by defining a simple version of your problem--how to evaluate a bunch of boolean expressions and find out if they all returned true. Just **AND**ing them together worked and was syntactically clean but revealed the need for better result reporting. So you wrote some really simpleminded code, chock-full of

duplication and error-prone idioms that reported the results the way you wanted.

The next step was to see if you could refactor the second version into something as clean as the former. You started with a standard refactoring technique of extracting some code into a function, `report-result`. Unfortunately, you could see that using `report-result` was going to be tedious and error-prone since you had to pass the test expression twice, once for the value and once as quoted data. So you wrote the `check` macro to automate the details of calling `report-result` correctly.

While writing `check`, you realized as long as you were generating code, you could make a single call to `check` to generate multiple calls to `report-result`, getting you back to a version of `test-+` about as concise as the original **`AND`** version.

At that point you had the `check` API nailed down, which allowed you to start mucking with how it worked on the inside. The next task was to fix `check` so the code it generated would return a boolean indicating whether all the test cases had passed. Rather than immediately hacking away at `check`, you paused to indulge in a little language design by fantasy. What if--you fantasized--there was already a non-short-circuiting **`AND`** construct. Then fixing `check` would be trivial. Returning from fantasyland you realized there was no such construct but that you could write one in a few lines. After writing `combine-results`, the fix to `check` was indeed trivial.

At that point all that was left was to make a few more improvements to the way you reported test results. Once you started making

changes to the test functions, you realized those functions represented a special category of function that deserved its own abstraction. So you wrote `deftest` to abstract the pattern of code that turns a regular function into a test function.

With `deftest` providing an abstraction barrier between the test definitions and the underlying machinery, you were able to enhance the result reporting without touching the test functions.

Now, with the basics of functions, variables, and macros mastered, and a little practical experience using them, you're ready to start exploring Common Lisp's rich standard library of functions and data types.

---

[1]This is for illustrative purposes only--obviously, writing test cases for built-in functions such as `+` is a bit silly, since if such basic things aren't working, the chances the tests will be running the way you expect is pretty slim. On the other hand, most Common Lisps are implemented largely in Common Lisp, so it's not crazy to imagine writing test suites in Common Lisp to test the standard library functions.

[2]Side effects can include such things as signaling errors; I'll discuss Common Lisp's error handling system in Chapter 19. You may, after reading that chapter, want to think about how to incorporate tests that check whether a function does or does not signal a particular error in certain situations.

[3]I'll discuss this and other **FORMAT** directives in more detail in Chapter 18.

[4]If `test-+` has been compiled--which may happen implicitly in certain Lisp implementations--you may need to reevaluate the definition of `test-+` to get the changed definition of `check` to affect the behavior of `test-+`. Interpreted code, on the other hand, typically expands macros anew each time the code is interpreted, allowing the effects of macro redefinitions to be seen immediately.

[5]You have to change the test to make it fail since you can't change the behavior of `+`.

[6]Though, again, if the test functions have been compiled, you'll have to recompile them after changing the macro.

[7]As you'll see in Chapter 12, **APPEND**ing to the end of a list isn't the most efficient way to build a list. But for now this is sufficient--as long as the test hierarchies aren't too deep, it should be fine. And if it becomes a problem, all you'll have to do is change the definition of `deftest`.

Copyright © 2003-2005, Peter Seibel

10. Numbers, Characters, and Strings

While functions, variables, macros, and 25 special operators provide the basic building blocks of the language itself, the building blocks of your programs will be the data structures you use. As Fred Brooks observed in *The Mythical Man-Month*, "Representation *is* the essence of programming."[1]

Common Lisp provides built-in support for most of the data types typically found in modern languages: numbers (integer, floating point, and complex), characters, strings, arrays (including multidimensional arrays), lists, hash tables, input and output streams, and an abstraction for portably representing filenames. Functions are also a first-class data type in Lisp--they can be stored in variables, passed as arguments, returned as return values, and created at runtime.

And these built-in types are just the beginning. They're defined in the language standard so programmers can count on them being available and because they tend to be easier to implement efficiently when tightly integrated with the rest of the implementation. But, as you'll see in later chapters, Common Lisp also provides several ways for you to define new data types, define operations on them, and integrate them with the built-in data types.

For now, however, you can start with the built-in data types. Because Lisp is a high-level language, the details of exactly how different data

types are implemented are largely hidden. From your point of view as a user of the language, the built-in data types are defined by the functions that operate on them. So to learn a data type, you just have to learn about the functions you can use with it. Additionally, most of the built-in data types have a special syntax that the Lisp reader understands and that the Lisp printer uses. That's why, for instance, you can write strings as `"foo"`; numbers as `123`, `1/23`, and `1.23`; and lists as `(a b c)`. I'll describe the syntax for different kinds of objects when I describe the functions for manipulating them.

In this chapter, I'll cover the built-in "scalar" data types: numbers, characters, and strings. Technically, strings aren't true scalars--a string is a sequence of characters, and you can access individual characters and manipulate strings with a function that operates on sequences. But I'll discuss strings here because most of the string-specific functions manipulate them as single values and also because of the close relation between several of the string functions and their character counterparts.

Numbers

Math, as Barbie says, is hard.[2] Common Lisp can't make the math part any easier, but it does tend to get in the way a lot less than other programming languages. That's not surprising given its mathematical heritage. Lisp was originally designed by a mathematician as a tool for studying mathematical functions. And one of the main projects of the MAC project at MIT was the Macsyma symbolic algebra system, written in Maclisp, one of Common Lisp's immediate predecessors. Additionally, Lisp has been used as a teaching language at places such as MIT where even the computer science professors cringe at the thought of telling their students that *10/4* = *2*, leading to Lisp's support for exact ratios. And at various times Lisp has been called upon to compete with FORTRAN in the high-performance numeric computing arena.

One of the reasons Lisp is a nice language for math is its numbers behave more like true mathematical numbers than the approximations of numbers that are easy to implement in finite computer hardware. For instance, integers in Common Lisp can be almost arbitrarily large rather than being limited by the size of a machine word.[3] And dividing two integers results in an exact ratio, not a truncated value. And since ratios are represented as pairs of arbitrarily sized integers, ratios can represent arbitrarily precise fractions.[4]

On the other hand, for high-performance numeric programming, you may be willing to trade the exactitude of rationals for the speed offered by using the hardware's underlying floating-point operations.

So, Common Lisp also offers several types of floating-point numbers, which are mapped by the implementation to the appropriate hardware-supported floating-point representations.[5] Floats are also used to represent the results of a computation whose true mathematical value would be an irrational number.

Finally, Common Lisp supports complex numbers--the numbers that result from doing things such as taking square roots and logarithms of negative numbers. The Common Lisp standard even goes so far as to specify the principal values and branch cuts for irrational and transcendental functions on the complex domain.

Numeric Literals

You can write numeric literals in a variety of ways; you saw a few examples in Chapter 4. However, it's important to keep in mind the division of labor between the Lisp reader and the Lisp evaluator--the reader is responsible for translating text into Lisp objects, and the Lisp evaluator then deals only with those objects. For a given number of a given type, there can be many different textual representations, all of which will be translated to the same object representation by the Lisp reader. For instance, you can write the integer 10 as `10`, `20/2`, `#xA`, or any of a number of other ways, but the reader will translate all these to the same object. When numbers are printed back out--say, at the REPL--they're printed in a canonical textual syntax that may be different from the syntax used to enter the number. For example:

```
CL-USER> 10
10
CL-USER> 20/2
10
CL-USER> #xa
10
```

The syntax for integer values is an optional sign (`+` or `-`) followed by one or more digits. Ratios are written as an optional sign and a sequence of digits, representing the numerator, a slash (`/`), and another sequence of digits representing the denominator. All rational numbers are "canonicalized" as they're read--that's why `10` and `20/2` are both read as the same number, as are `3/4` and `6/8`. Rationals are printed in "reduced" form--integer values are printed in integer syntax and ratios with the numerator and denominator reduced to lowest terms.

It's also possible to write rationals in bases other than 10. If preceded by `#B` or `#b`, a rational literal is read as a binary number with `0` and `1` as the only legal digits. An `#O` or `#o` indicates an octal number (legal digits `0`-`7`), and `#X` or `#x` indicates hexadecimal (legal digits `0`-`F` or `0`-`f`). You can write rationals in other bases from 2 to 36 with `#nR` where *n* is the base (always written in decimal). Additional "digits" beyond 9 are taken from the letters `A`-`Z` or `a`-`z`. Note that these radix indicators apply to the whole rational--it's not possible to write a ratio with the numerator in one base and denominator in another. Also, you can write integer values, but not ratios, as decimal digits terminated with a decimal point.[6] Some examples of rationals, with their canonical, decimal representation are as follows:

```
123                             ==> 123
+123                            ==> 123
-123                            ==> -123
123.                            ==> 123
2/3                             ==> 2/3
-2/3                            ==> -2/3
4/6                             ==> 2/3
6/3                             ==> 2
#b10101                         ==> 21
#b1010/1011                     ==> 10/11
#o777                           ==> 511
#xDADA                          ==> 56026
#36rABCDEFGHIJKLMNOPQRSTUVWXYZ ==> 833750385473041524105037713581125926783 5
```

You can also write floating-point numbers in a variety of ways. Unlike rational numbers, the syntax used to notate a floating-point number can affect the actual type of number read. Common Lisp defines four subtypes of floating-point number: short, single, double, and long. Each subtype can use a different number of bits in its representation, which means each subtype can represent values spanning a different range and with different precision. More bits gives a wider range and more precision.[7]

The basic format for floating-point numbers is an optional sign followed by a nonempty sequence of decimal digits possibly with an embedded decimal point. This sequence can be followed by an exponent marker for "computerized scientific notation."[8] The exponent marker consists of a single letter followed by an optional sign and a sequence of digits, which are interpreted as the power of ten by which the number before the exponent marker should be multiplied. The letter does double duty: it marks the beginning of the exponent and indicates what floating- point representation should be used for the number. The exponent markers *s*, *f*, *d*, *l* (and their uppercase equivalents) indicate short, single, double, and long floats, respectively. The letter *e* indicates that the default representation (initially single-float) should be used.

Numbers with no exponent marker are read in the default representation and must contain a decimal point followed by at least one digit to distinguish them from integers. The digits in a floating-point number are always treated as base 10 digits--the `#B`, `#X`, `#O`, and `#R` syntaxes work only with rationals. The following are some example floating-point numbers along with their canonical representation:

```
1.0      ==> 1.0
1e0      ==> 1.0
1d0      ==> 1.0d0
123.0    ==> 123.0
123e0    ==> 123.0
0.123    ==> 0.123
.123     ==> 0.123
123e-3   ==> 0.123
123E-3   ==> 0.123
0.123e20 ==> 1.23e+19
123d23   ==> 1.23d+25
```

Finally, complex numbers are written in their own syntax, namely, `#C` or `#c` followed by a list of two real numbers representing the real and imaginary part of the complex number. There are actually five kinds of complex numbers because the real and imaginary parts must either both be rational or both be the same kind of floating-point number.

But you can write them however you want--if a complex is written with one rational and one floating-point part, the rational is converted to a float of the appropriate representation. Similarly, if the real and imaginary parts are both floats of different representations, the one in the smaller representation will be *upgraded*.

However, no complex numbers have a rational real component and a zero imaginary part--since such values are, mathematically speaking, rational, they're represented by the appropriate rational value. The same mathematical argument could be made for complex numbers with floating-point components, but for those complex types a number with a zero imaginary part is always a different object than the floating-point number representing the real component. Here are some examples of numbers written the complex number syntax:

```
#c(2      1)    ==> #c(2 1)
#c(2/3  3/4)    ==> #c(2/3 3/4)
#c(2    1.0)    ==> #c(2.0 1.0)
#c(2.0  1.0d0)  ==> #c(2.0d0 1.0d0)
#c(1/2  1.0)    ==> #c(0.5 1.0)
#c(3      0)    ==> 3
#c(3.0  0.0)    ==> #c(3.0 0.0)
#c(1/2    0)    ==> 1/2
#c(-6/3   0)    ==> -2
```

Basic Math

The basic arithmetic operations--addition, subtraction, multiplication, and division--are supported for all the different kinds of Lisp numbers with the functions `+`, `-`, `*`, and `/`. Calling any of these functions with more than two arguments is equivalent to calling the same function on the first two arguments and then calling it again on the resulting value and the rest of the arguments. For example, `(+ 1 2 3)` is equivalent to `(+ (+ 1 2) 3)`. With only one argument, `+` and `*` return the value; `-` returns its negation and `/` its reciprocal.[9]

```
(+ 1 2)              ==> 3
(+ 1 2 3)            ==> 6
(+ 10.0 3.0)         ==> 13.0
(+ #c(1 2) #c(3 4))  ==> #c(4 6)
(- 5 4)              ==> 1
(- 2)                ==> -2
(- 10 3 5)           ==> 2
(* 2 3)              ==> 6
(* 2 3 4)            ==> 24
(/ 10 5)             ==> 2
(/ 10 5 2)           ==> 1
(/ 2 3)              ==> 2/3
(/ 4)                ==> 1/4
```

If all the arguments are the same type of number (rational, floating point, or complex), the result will be the same type except in the case where the result of an operation on complex numbers with rational components yields a number with a zero imaginary part, in which case the result will be a rational. However, floating-point and complex numbers are *contagious*--if all the arguments are reals but one or more are floating-point numbers, the other arguments are converted to the nearest floating-point value in a "largest" floating-point representation of the actual floating-point arguments. Floating-point numbers in a "smaller" representation are also converted to the

larger representation. Similarly, if any of the arguments are complex, any real arguments are converted to the complex equivalents.

```
(+ 1 2.0)             ==> 3.0
(/ 2 3.0)             ==> 0.6666667
(+ #c(1 2) 3)         ==> #c(4 2)
(+ #c(1 2) 3/2)       ==> #c(5/2 2)
(+ #c(1 1) #c(2 -1))  ==> 3
```

Because `/` doesn't truncate, Common Lisp provides four flavors of truncating and rounding for converting a real number (rational or floating point) to an integer: **FLOOR** truncates toward negative infinity, returning the largest integer less than or equal to the argument. **CEILING** truncates toward positive infinity, returning the smallest integer greater than or equal to the argument. **TRUNCATE** truncates toward zero, making it equivalent to **FLOOR** for positive arguments and to **CEILING** for negative arguments. And **ROUND** rounds to the nearest integer. If the argument is exactly halfway between two integers, it rounds to the nearest even integer.

Two related functions are **MOD** and **REM**, which return the modulus and remainder of a truncating division on real numbers. These two functions are related to the **FLOOR** and **TRUNCATE** functions as follows:

```
(+ (* (floor    (/ x y)) y) (mod x y)) === x
(+ (* (truncate (/ x y)) y) (rem x y)) === x
```

Thus, for positive quotients they're equivalent, but for negative quotients they produce different results.[10]

The functions `1+` and `1-` provide a shorthand way to express adding and subtracting one from a number. Note that these are different from the macros **INCF** and **DECF**. `1+` and `1-` are just functions that

return a new value, but **INCF** and **DECF** modify a place. The following equivalences show the relation between **INCF**/**DECF**, **1+**/**1-**, and **+**/**-**:

```
(incf x)    === (setf x (1+ x)) === (setf x (+ x 1))
(decf x)    === (setf x (1- x)) === (setf x (- x 1))
(incf x 10) === (setf x (+ x 10))
(decf x 10) === (setf x (- x 10))
```

Numeric Comparisons

The function = is the numeric equality predicate. It compares numbers by mathematical value, ignoring differences in type. Thus, = will consider mathematically equivalent values of different types equivalent while the generic equality predicate **EQL** would consider them inequivalent because of the difference in type. (The generic equality predicate **EQUALP**, however, uses = to compare numbers.) If it's called with more than two arguments, it returns true only if they all have the same value. Thus:

```
(= 1 1)                         ==> T
(= 10 20/2)                     ==> T
(= 1 1.0 #c(1.0 0.0) #c(1 0))   ==> T
```

The /= function, conversely, returns true only if *all* its arguments are different values.

```
(/= 1 1)         ==> NIL
(/= 1 2)         ==> T
(/= 1 2 3)       ==> T
(/= 1 2 3 1)     ==> NIL
(/= 1 2 3 1.0)   ==> NIL
```

The functions <, >, <=, and >= order rationals and floating-point numbers (in other words, the real numbers.) Like = and /=, these functions can be called with more than two arguments, in which case each argument is compared to the argument to its right.

```
(< 2 3)        ==> T
(> 2 3)        ==> NIL
(> 3 2)        ==> T
(< 2 3 4)      ==> T
(< 2 3 3)      ==> NIL
(<= 2 3 3)     ==> T
(<= 2 3 3 4)   ==> T
(<= 2 3 4 3)   ==> NIL
```

To pick out the smallest or largest of several numbers, you can use the function **MIN** or **MAX**, which takes any number of real number arguments and returns the minimum or maximum value.

```
(max 10 11)    ==> 11
(min -12 -10)  ==> -12
(max -1 2 -3)  ==> 2
```

Some other handy functions are **ZEROP**, **MINUSP**, and **PLUSP**, which test whether a single real number is equal to, less than, or greater than zero. Two other predicates, **EVENP** and **ODDP**, test whether a single integer argument is even or odd. The *P* suffix on the names of these functions is a standard naming convention for predicate functions, functions that test some condition and return a boolean.

Higher Math

The functions you've seen so far are the beginning of the built-in mathematical functions. Lisp also supports logarithms: `LOG`; exponentiation: `EXP` and `EXPT`; the basic trigonometric functions: `SIN`, `COS`, and `TAN`; their inverses: `ASIN`, `ACOS`, and `ATAN`; hyperbolic functions: `SINH`, `COSH`, and `TANH`; and their inverses: `ASINH`, `ACOSH`, and `ATANH`. It also provides functions to get at the individual bits of an integer and to extract the parts of a ratio or a complex number. For a complete list, see any Common Lisp reference.

Characters

Common Lisp characters are a distinct type of object from numbers. That's as it should be--characters are *not* numbers, and languages that treat them as if they are tend to run into problems when character encodings change, say, from 8-bit ASCII to 21-bit Unicode.[11] Because the Common Lisp standard didn't mandate a particular representation for characters, today several Lisp implementations use Unicode as their "native" character encoding despite Unicode being only a gleam in a standards body's eye at the time Common Lisp's own standardization was being wrapped up.

The read syntax for characters objects is simple: `#\` followed by the desired character. Thus, `#\x` is the character `x`. Any character can be used after the `#\`, including otherwise special characters such as `"`, `(`, and whitespace. However, writing whitespace characters this way isn't very (human) readable; an alternative syntax for certain characters is `#\` followed by the character's name. Exactly what names are supported depends on the character set and on the Lisp implementation, but all implementations support the names *Space* and *Newline*. Thus, you should write `#\Space` instead of `#\ `, though the latter is technically legal. Other semistandard names (that implementations must use if the character set has the appropriate characters) are *Tab*, *Page*, *Rubout*, *Linefeed*, *Return*, and *Backspace*.

Character Comparisons

The main thing you can do with characters, other than putting them into strings (which I'll get to later in this chapter), is to compare them with other characters. Since characters aren't numbers, you can't use the numeric comparison functions, such as < and >. Instead, two sets of functions provide character-specific analogs to the numeric comparators; one set is case-sensitive and the other case-insensitive.

The case-sensitive analog to the numeric = is the function **CHAR=**. Like =, **CHAR=** can take any number of arguments and returns true only if they're all the same character. The case- insensitive version is **CHAR-EQUAL**.

The rest of the character comparators follow this same naming scheme: the case-sensitive comparators are named by prepending the analogous numeric comparator with CHAR; the case-insensitive versions spell out the comparator name, separated from the CHAR with a hyphen. Note, however, that <= and >= are "spelled out" with the logical equivalents **NOT-GREATERP** and **NOT-LESSP** rather than the more verbose LESSP-OR-EQUALP and GREATERP-OR-EQUALP. Like their numeric counterparts, all these functions can take one or more arguments. Table 10-1 summarizes the relation between the numeric and character comparison functions.

Table 10-1. Character Comparison Functions

Numeric Analog	Case-Sensitive	Case-Insensitive
=	CHAR=	CHAR-EQUAL
/=	CHAR/=	CHAR-NOT-EQUAL

<	CHAR<	CHAR-LESSP
>	CHAR>	CHAR-GREATERP
<=	CHAR<=	CHAR-NOT-GREATERP
>=	CHAR>=	CHAR-NOT-LESSP

Other functions that deal with characters provide functions for, among other things, testing whether a given character is alphabetic or a digit character, testing the case of a character, obtaining a corresponding character in a different case, and translating between numeric values representing character codes and actual character objects. Again, for complete details, see your favorite Common Lisp reference.

Strings

As mentioned earlier, strings in Common Lisp are really a composite data type, namely, a one-dimensional array of characters. Consequently, I'll cover many of the things you can do with strings in the next chapter when I discuss the many functions for manipulating sequences, of which strings are just one type. But strings also have their own literal syntax and a library of functions for performing string-specific operations. I'll discuss these aspects of strings in this chapter and leave the others for Chapter 11.

As you've seen, literal strings are written enclosed in double quotes. You can include any character supported by the character set in a literal string except double quote (`"`) and backslash (\). And you can include these two as well if you escape them with a backslash. In fact, backslash always escapes the next character, whatever it is, though this isn't necessary for any character except for `"` and itself. Table 10-2 shows how various literal strings will be read by the Lisp reader.

Table 10-2. Literal Strings

Literal	Contents	Comment
`"foobar"`	foobar	Plain string.
`"foo\"bar"`	foo"bar	The backslash escapes quote.
`"foo\\bar"`	foo\bar	The first backslash escapes second backslash.
`"\"foobar\""`	"foobar"	The backslashes escape quotes.
`"foo\bar"`	foobar	The backslash "escapes" *b*

Note that the REPL will ordinarily print strings in readable form, adding the enclosing quotation marks and any necessary escaping backslashes, so if you want to see the actual contents of a string, you need to use function such as **FORMAT** designed to print human-readable output. For example, here's what you see if you type a string containing an embedded quotation mark at the REPL:

```
CL-USER> "foo\"bar"
"foo\"bar"
```

FORMAT, on the other hand, will show you the actual string contents:[12]

```
CL-USER> (format t "foo\"bar")
foo"bar
NIL
```

String Comparisons

You can compare strings using a set of functions that follow the same naming convention as the character comparison functions except with `STRING` as the prefix rather than `CHAR` (see Table 10-3).

Table 10-3. String Comparison Functions

Numeric Analog	Case-Sensitive	Case-Insensitive
=	STRING=	STRING-EQUAL
/=	STRING/=	STRING-NOT-EQUAL
<	STRING<	STRING-LESSP
>	STRING>	STRING-GREATERP
<=	STRING<=	STRING-NOT-GREATERP
>=	STRING>=	STRING-NOT-LESSP

However, unlike the character and number comparators, the string comparators can compare only two strings. That's because they also take keyword arguments that allow you to restrict the comparison to a substring of either or both strings. The arguments--`:start1`, `:end1`, `:start2`, and `:end2`--specify the starting (inclusive) and ending (exclusive) indices of substrings in the first and second string arguments. Thus, the following:

```
(string= "foobarbaz" "quuxbarfoo" :start1 3 :end1 6 :start2 4 :end2 7)
```

compares the substring "bar" in the two arguments and returns true. The `:end1` and `:end2` arguments can be **`NIL`** (or the keyword argument omitted altogether) to indicate that the corresponding substring extends to the end of the string.

The comparators that return true when their arguments differ--that is, all of them except **`STRING=`** and **`STRING-EQUAL`**--return the index in the

first string where the mismatch was detected.

```
(string/= "lisp" "lissome") ==> 3
```

If the first string is a prefix of the second, the return value will be the length of the first string, that is, one greater than the largest valid index into the string.

```
(string< "lisp" "lisper") ==> 4
```

When comparing substrings, the resulting value is still an index into the string as a whole. For instance, the following compares the substrings "bar" and "baz" but returns 5 because that's the index of the *r* in the first string:

```
(string< "foobar" "abaz" :start1 3 :start2 1) ==> 5   ; N.B. not 2
```

Other string functions allow you to convert the case of strings and trim characters from one or both ends of a string. And, as I mentioned previously, since strings are really a kind of sequence, all the sequence functions I'll discuss in the next chapter can be used with strings. For instance, you can discover the length of a string with the **LENGTH** function and can get and set individual characters of a string with the generic sequence element accessor function, **ELT**, or the generic array element accessor function, **AREF**. Or you can use the string-specific accessor, **CHAR**. But those functions, and others, are the topic of the next chapter, so let's move on.

---

[1]Fred Brooks, *The Mythical Man-Month*, 20th Anniversary Edition (Boston: Addison-Wesley, 1995), p. 103. Emphasis in original.

[2]Mattel's Teen Talk Barbie

[3]Obviously, the size of a number that can be represented on a computer with finite memory is still limited in practice; furthermore, the actual representation of *bignums* used in a particular Common Lisp implementation may place other limits on the size of number that can be represented. But these limits are going to be well beyond "astronomically" large numbers. For instance, the number of atoms in the universe is estimated to be less than 2^269; current Common Lisp implementations can easily handle numbers up to and beyond 2^262144.

[4]Folks interested in using Common Lisp for intensive numeric computation should note that a naive comparison of the performance of numeric code in Common Lisp and languages such as C or FORTRAN will probably show Common Lisp to be much slower. This is because something as simple as `(+ a b)` in Common Lisp is doing a lot more than the seemingly equivalent `a + b` in one of those languages. Because of Lisp's dynamic typing and support for things such as arbitrary precision rationals and complex numbers, a seemingly simple addition is doing a lot more than an addition of two numbers that are known to be represented by machine words. However, you can use declarations to give Common Lisp information about the types of numbers you're using that will enable it to generate code that does only as much work as the code that would be generated by a C or FORTRAN compiler. Tuning numeric code for this kind of performance is beyond the scope of this book, but it's certainly possible.

[5]While the standard doesn't require it, many Common Lisp implementations support the IEEE standard for floating-point arithmetic, *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/ IEEE Std 754-1985* (Institute of Electrical and Electronics Engineers, 1985).

[6]It's also possible to change the default base the reader uses for numbers without a specific radix marker by changing the value of the global variable **`*READ-BASE*`**. However, it's not clear that's the path to anything other than complete insanity.

[7]Since the purpose of floating-point numbers is to make efficient use of floating-point hardware, each Lisp implementation is allowed to map these four subtypes onto the native floating-point types as appropriate. If the hardware supports fewer than four distinct representations, one or more of the types may be equivalent.

[8]"Computerized scientific notation" is in scare quotes because, while commonly used in computer languages since the days of FORTRAN, it's actually quite different from real scientific notation. In particular, something like `1.0e4` means `10000.0`, but in true scientific notation that would be written as 1.0 x 10^4. And to further confuse matters, in true scientific

notation the letter *e* stands for the base of the natural logarithm, so something like 1.0 x *e*^4, while superficially similar to `1.0e4`, is a completely different value, approximately 54.6.

[9]For mathematical consistency, `+` and `*` can also be called with no arguments, in which case they return the appropriate identity: 0 for `+` and 1 for `*`.

[10]Roughly speaking, **MOD** is equivalent to the `%` operator in Perl and Python, and **REM** is equivalent to the % in C and Java. (Technically, the exact behavior of % in C wasn't specified until the C99 standard.)

[11]Even Java, which was designed from the beginning to use Unicode characters on the theory that Unicode was the going to be *the* character encoding of the future, has run into trouble since Java characters are defined to be a 16-bit quantity and the Unicode 3.1 standard extended the range of the Unicode character set to require a 21-bit representation. Ooops.

[12]Note, however, that not all literal strings can be printed by passing them as the second argument to **FORMAT** since certain sequences of characters have a special meaning to **FORMAT**. To safely print an arbitrary string--say, the value of a variable s--with **FORMAT** you should write (format t "~a" s).

Copyright © 2003-2005, Peter Seibel

11. Collections

Like most programming languages, Common Lisp provides standard data types that collect multiple values into a single object. Every language slices up the collection problem a little bit differently, but the basic collection types usually boil down to an integer-indexed array type and a table type that can be used to map more or less arbitrary keys to values. The former are variously called *arrays*, *lists*, or *tuples*; the latter go by the names *hash tables*, *associative arrays*, *maps*, and *dictionaries*.

Lisp is, of course, famous for its list data structure, and most Lisp books, following the ontogeny-recapitulates-phylogeny principle of language instruction, start their discussion of Lisp's collections with lists. However, that approach often leads readers to the mistaken conclusion that lists are Lisp's *only* collection type. To make matters worse, because Lisp's lists are such a flexible data structure, it *is* possible to use them for many of the things arrays and hash tables are used for in other languages. But it's a mistake to focus too much on lists; while they're a crucial data structure for representing Lisp code as Lisp data, in many situations other data structures are more appropriate.

To keep lists from stealing the show, in this chapter I'll focus on Common Lisp's other collection types: vectors and hash tables.[1] However, vectors and lists share enough characteristics that Common Lisp treats them both as subtypes of a more general

abstraction, the sequence. Thus, you can use many of the functions I'll discuss in this chapter with both vectors and lists.

Vectors

Vectors are Common Lisp's basic integer-indexed collection, and they come in two flavors. Fixed-size vectors are a lot like arrays in a language such as Java: a thin veneer over a chunk of contiguous memory that holds the vector's elements.[2] Resizable vectors, on the other hand, are more like arrays in Perl or Ruby, lists in Python, or the ArrayList class in Java: they abstract the actual storage, allowing the vector to grow and shrink as elements are added and removed.

You can make fixed-size vectors containing specific values with the function **VECTOR**, which takes any number of arguments and returns a freshly allocated fixed-size vector containing those arguments.

```
(vector)     ==> #()
(vector 1)   ==> #(1)
(vector 1 2) ==> #(1 2)
```

The `#(...)` syntax is the literal notation for vectors used by the Lisp printer and reader. This syntax allows you to save and restore vectors by **PRINT**ing them out and **READ**ing them back in. You can use the `#(...)` syntax to include literal vectors in your code, but as the effects of modifying literal objects aren't defined, you should always use **VECTOR** or the more general function **MAKE-ARRAY** to create vectors you plan to modify.

MAKE-ARRAY is more general than **VECTOR** since you can use it to create arrays of any dimensionality as well as both fixed-size and resizable vectors. The one required argument to **MAKE-ARRAY** is a list containing the dimensions of the array. Since a vector is a one-dimensional array, this list will contain one number, the size of the

vector. As a convenience, **MAKE-ARRAY** will also accept a plain number in the place of a one-item list. With no other arguments, **MAKE-ARRAY** will create a vector with uninitialized elements that must be set before they can be accessed.[3] To create a vector with the elements all set to a particular value, you can pass an `:initial-element` argument. Thus, to make a five-element vector with its elements initialized to **NIL**, you can write the following:

```
(make-array 5 :initial-element nil) ==> #(NIL NIL NIL NIL NIL)
```

MAKE-ARRAY is also the function to use to make a resizable vector. A resizable vector is a slightly more complicated object than a fixed-size vector; in addition to keeping track of the memory used to hold the elements and the number of slots available, a resizable vector also keeps track of the number of elements actually stored in the vector. This number is stored in the vector's *fill pointer*, so called because it's the index of the next position to be filled when you add an element to the vector.

To make a vector with a fill pointer, you pass **MAKE-ARRAY** a `:fill-pointer` argument. For instance, the following call to **MAKE-ARRAY** makes a vector with room for five elements; but it looks empty because the fill pointer is zero:

```
(make-array 5 :fill-pointer 0) ==> #()
```

To add an element to the end of a resizable vector, you can use the function **VECTOR-PUSH**. It adds the element at the current value of the fill pointer and then increments the fill pointer by one, returning the index where the new element was added. The function **VECTOR-POP**

returns the most recently pushed item, decrementing the fill pointer in the process.

```
(defparameter *x* (make-array 5 :fill-pointer 0))

(vector-push 'a *x*) ==> 0
*x*                  ==> #(A)
(vector-push 'b *x*) ==> 1
*x*                  ==> #(A B)
(vector-push 'c *x*) ==> 2
*x*                  ==> #(A B C)
(vector-pop *x*)     ==> C
*x*                  ==> #(A B)
(vector-pop *x*)     ==> B
*x*                  ==> #(A)
(vector-pop *x*)     ==> A
*x*                  ==> #()
```

However, even a vector with a fill pointer isn't completely resizable. The vector `*x*` can hold at most five elements. To make an arbitrarily resizable vector, you need to pass **`MAKE-ARRAY`** another keyword argument: `:adjustable`.

```
(make-array 5 :fill-pointer 0 :adjustable t) ==> #()
```

This call makes an *adjustable* vector whose underlying memory can be resized as needed. To add elements to an adjustable vector, you use **`VECTOR-PUSH-EXTEND`**, which works just like **`VECTOR-PUSH`** except it will automatically expand the array if you try to push an element onto a full vector--one whose fill pointer is equal to the size of the underlying storage.[4]

Subtypes of Vector

All the vectors you've dealt with so far have been *general* vectors that can hold any type of object. It's also possible to create *specialized* vectors that are restricted to holding certain types of elements. One reason to use specialized vectors is they may be stored more compactly and can provide slightly faster access to their elements than general vectors. However, for the moment let's focus on a couple kinds of specialized vectors that are important data types in their own right.

One of these you've seen already--strings are vectors specialized to hold characters. Strings are important enough to get their own read/print syntax (double quotes) and the set of string-specific functions I discussed in the previous chapter. But because they're also vectors, all the functions I'll discuss in the next few sections that take vector arguments can also be used with strings. These functions will fill out the string library with functions for things such as searching a string for a substring, finding occurrences of a character within a string, and more.

Literal strings, such as `"foo"`, are like literal vectors written with the `#()` syntax--their size is fixed, and they must not be modified. However, you can use **`MAKE-ARRAY`** to make resizable strings by adding another keyword argument, `:element-type`. This argument takes a *type* descriptor. I won't discuss all the possible type descriptors you can use here; for now it's enough to know you can create a string by passing the symbol **`CHARACTER`** as the `:element-type` argument. Note that you need to quote the symbol to

prevent it from being treated as a variable name. For example, to make an initially empty but resizable string, you can write this:

```
(make-array 5 :fill-pointer 0 :adjustable t :element-type 'character)  ""
```

Bit vectors--vectors whose elements are all zeros or ones--also get some special treatment. They have a special read/print syntax that looks like `#*00001111` and a fairly large library of functions, which I won't discuss, for performing bit-twiddling operations such as "anding" together two bit arrays. The type descriptor to pass as the `:element-type` to create a bit vector is the symbol **`BIT`**.

Vectors As Sequences

As mentioned earlier, vectors and lists are the two concrete subtypes of the abstract type *sequence*. All the functions I'll discuss in the next few sections are sequence functions; in addition to being applicable to vectors--both general and specialized--they can also be used with lists.

The two most basic sequence functions are **LENGTH**, which returns the length of a sequence, and **ELT**, which allows you to access individual elements via an integer index. **LENGTH** takes a sequence as its only argument and returns the number of elements it contains. For vectors with a fill pointer, this will be the value of the fill pointer. **ELT**, short for *element*, takes a sequence and an integer index between zero (inclusive) and the length of the sequence (exclusive) and returns the corresponding element. **ELT** will signal an error if the index is out of bounds. Like **LENGTH**, **ELT** treats a vector with a fill pointer as having the length specified by the fill pointer.

```
(defparameter *x* (vector 1 2 3))

(length *x*) ==> 3
(elt *x* 0)  ==> 1
(elt *x* 1)  ==> 2
(elt *x* 2)  ==> 3
(elt *x* 3)  ==> error
```

ELT is also a **SETF**able place, so you can set the value of a particular element like this:

```
(setf (elt *x* 0) 10)

*x* ==> #(10 2 3)
```

Sequence Iterating Functions

While in theory all operations on sequences boil down to some combination of **LENGTH**, **ELT**, and **SETF** of **ELT** operations, Common Lisp provides a large library of sequence functions.

One group of sequence functions allows you to express certain operations on sequences such as finding or filtering specific elements without writing explicit loops. Table 11-1 summarizes them.

Table 11-1.Basic Sequence Functions

Name	Required Arguments	Returns
COUNT	Item and sequence	Number of times item appears in sequence
FIND	Item and sequence	Item or **NIL**
POSITION	Item and sequence	Index into sequence or **NIL**
REMOVE	Item and sequence	Sequence with instances of item removed
SUBSTITUTE	New item, item, and sequence	Sequence with instances of item replaced with new item

Here are some simple examples of how to use these functions:

```
(count 1 #(1 2 1 2 3 1 2 3 4))         ==> 3
(remove 1 #(1 2 1 2 3 1 2 3 4))        ==> #(2 2 3 2 3 4)
(remove 1 '(1 2 1 2 3 1 2 3 4))        ==> (2 2 3 2 3 4)
(remove #\a "foobarbaz")               ==> "foobrbz"
(substitute 10 1 #(1 2 1 2 3 1 2 3 4)) ==> #(10 2 10 2 3 10 2 3 4)
(substitute 10 1 '(1 2 1 2 3 1 2 3 4)) ==> (10 2 10 2 3 10 2 3 4)
(substitute #\x #\b "foobarbaz")       ==> "fooxarxaz"
(find 1 #(1 2 1 2 3 1 2 3 4))          ==> 1
(find 10 #(1 2 1 2 3 1 2 3 4))         ==> NIL
(position 1 #(1 2 1 2 3 1 2 3 4))      ==> 0
```

Note how **REMOVE** and **SUBSTITUTE** always return a sequence of the same type as their sequence argument.

You can modify the behavior of these five functions in a variety of ways using keyword arguments. For instance, these functions, by default, look for elements in the sequence that are the same object as the item argument. You can change this in two ways: First, you can use the `:test` keyword to pass a function that accepts two arguments and returns a boolean. If provided, it will be used to compare *item* to each element instead of the default object equality test, **EQL**.[5] Second, with the `:key` keyword you can pass a one-argument function to be called on each element of the sequence to extract a *key* value, which will then be compared to the item in the place of the element itself. Note, however, that functions such as **FIND** that return elements of the sequence continue to return the actual element, not just the extracted key.

```
(count "foo" #("foo" "bar" "baz") :test #'string=)    ==> 1
(find 'c #((a 10) (b 20) (c 30) (d 40)) :key #'first) ==> (C 30)
```

To limit the effects of these functions to a particular subsequence of the sequence argument, you can provide bounding indices with `:start` and `:end` arguments. Passing **NIL** for `:end` or omitting it is the same as specifying the length of the sequence.[6]

If a non-**NIL** `:from-end` argument is provided, then the elements of the sequence will be examined in reverse order. By itself `:from-end` can affect the results of only **FIND** and **POSITION**. For instance:

```
(find 'a #((a 10) (b 20) (a 30) (b 40)) :key #'first)             ==> (A 10)
(find 'a #((a 10) (b 20) (a 30) (b 40)) :key #'first :from-end t) ==> (A 30)
```

However, the `:from-end` argument can affect **REMOVE** and **SUBSTITUTE** in conjunction with another keyword parameter, `:count`, that's used to specify how many elements to remove or substitute. If you specify a

`:count` lower than the number of matching elements, then it obviously matters which end you start from:

```
(remove #\a "foobarbaz" :count 1)             ==> "foobrbaz"
(remove #\a "foobarbaz" :count 1 :from-end t) ==> "foobarbz"
```

And while `:from-end` can't change the results of the **COUNT** function, it does affect the order the elements are passed to any `:test` and `:key` functions, which could possibly have side effects. For example:

```
CL-USER> (defparameter *v* #((a 10) (b 20) (a 30) (b 40)))
*V*
CL-USER> (defun verbose-first (x) (format t "Looking at ~s~%" x) (first x))
VERBOSE-FIRST
CL-USER> (count 'a *v* :key #'verbose-first)
Looking at (A 10)
Looking at (B 20)
Looking at (A 30)
Looking at (B 40)
2
CL-USER> (count 'a *v* :key #'verbose-first :from-end t)
Looking at (B 40)
Looking at (A 30)
Looking at (B 20)
Looking at (A 10)
2
```

Table 11-2 summarizes these arguments.

Table 11-2. Standard Sequence Function Keyword Arguments

Argument	Meaning	Default
`:test`	Two-argument function used to compare item (or value extracted by `:key` function) to element.	**EQL**
`:key`	One-argument function to extract key value from actual sequence element. **NIL** means use element as is.	**NIL**
`:start`	Starting index (inclusive) of subsequence.	0
`:end`	Ending index (exclusive) of subsequence. **NIL** indicates end of sequence.	**NIL**
`:from-end`	If true, the sequence will be traversed in reverse order, from end to start.	**NIL**
`:count`	Number indicating the number of elements to remove or substitute or **NIL** to indicate all (**REMOVE** and **SUBSTITUTE** only).	**NIL**

Higher-Order Function Variants

For each of the functions just discussed, Common Lisp provides two *higher-order function* variants that, in the place of the item argument, take a function to be called on each element of the sequence. One set of variants are named the same as the basic function with an `-IF` appended. These functions count, find, remove, and substitute elements of the sequence for which the function argument returns true. The other set of variants are named with an `-IF-NOT` suffix and count, find, remove, and substitute elements for which the function argument does *not* return true.

```
(count-if #'evenp #(1 2 3 4 5))         ==> 2

(count-if-not #'evenp #(1 2 3 4 5))     ==> 3

(position-if #'digit-char-p "abcd0001") ==> 4

(remove-if-not #'(lambda (x) (char= (elt x 0) #\f))
  #("foo" "bar" "baz" "foom")) ==> #("foo" "foom")
```

According to the language standard, the **`-IF-NOT`** variants are deprecated. However, that deprecation is generally considered to have itself been ill-advised. If the standard is ever revised, it's more likely the deprecation will be removed than the **`-IF-NOT`** functions. For one thing, the **`REMOVE-IF-NOT`** variant is probably used more often than **`REMOVE-IF`**. Despite its negative-sounding name, **`REMOVE-IF-NOT`** is actually the positive variant--it returns the elements that *do* satisfy the predicate. [7]

The `-IF` and `-IF-NOT` variants accept all the same keyword arguments as their vanilla counterparts except for `:test`, which isn't needed since the main argument is already a function.[8] With a `:key`

argument, the value extracted by the `:key` function is passed to the function instead of the actual element.

```
(count-if #'evenp #((1 a) (2 b) (3 c) (4 d) (5 e)) :key #'first)     ==> 2

(count-if-not #'evenp #((1 a) (2 b) (3 c) (4 d) (5 e)) :key #'first) ==> 3

(remove-if-not #'alpha-char-p
  #("foo" "bar" "1baz") :key #'(lambda (x) (elt x 0))) ==> #("foo" "bar")
```

The **REMOVE** family of functions also support a fourth variant, **REMOVE-DUPLICATES**, that has only one required argument, a sequence, from which it removes all but one instance of each duplicated element. It takes the same keyword arguments as **REMOVE**, except for `:count`, since it always removes all duplicates.

```
(remove-duplicates #(1 2 1 2 3 1 2 3 4)) ==> #(1 2 3 4)
```

Whole Sequence Manipulations

A handful of functions perform operations on a whole sequence (or sequences) at a time. These tend to be simpler than the other functions I've described so far. For instance, COPY-SEQ and REVERSE each take a single argument, a sequence, and each returns a new sequence of the same type. The sequence returned by COPY-SEQ contains the same elements as its argument while the sequence returned by REVERSE contains the same elements but in reverse order. Note that neither function copies the elements themselves--only the returned sequence is a new object.

The CONCATENATE function creates a new sequence containing the concatenation of any number of sequences. However, unlike REVERSE and COPY-SEQ, which simply return a sequence of the same type as their single argument, CONCATENATE must be told explicitly what kind of sequence to produce in case the arguments are of different types. Its first argument is a type descriptor, like the `:element-type` argument to MAKE-ARRAY. In this case, the type descriptors you'll most likely use are the symbols VECTOR, LIST, or STRING.[9] For example:

```
(concatenate 'vector #(1 2 3) '(4 5 6))    ==> #(1 2 3 4 5 6)
(concatenate 'list #(1 2 3) '(4 5 6))      ==> (1 2 3 4 5 6)
(concatenate 'string "abc" '(#\d #\e #\f)) ==> "abcdef"
```

Sorting and Merging

The functions **SORT** and **STABLE-SORT** provide two ways of sorting a sequence. They both take a sequence and a two-argument predicate and return a sorted version of the sequence.

```
(sort (vector "foo" "bar" "baz") #'string<) ==> #("bar" "baz" "foo")
```

The difference is that **STABLE-SORT** is guaranteed to not reorder any elements considered equivalent by the predicate while **SORT** guarantees only that the result is sorted and may reorder equivalent elements.

Both these functions are examples of what are called *destructive* functions. Destructive functions are allowed--typically for reasons of efficiency--to modify their arguments in more or less arbitrary ways. This has two implications: one, you should always do something with the return value of these functions (such as assign it to a variable or pass it to another function), and, two, unless you're done with the object you're passing to the destructive function, you should pass a copy instead. I'll say more about destructive functions in the next chapter.

Typically you won't care about the unsorted version of a sequence after you've sorted it, so it makes sense to allow **SORT** and **STABLE-SORT** to destroy the sequence in the course of sorting it. But it does mean you need to remember to write the following:[10]

```
(setf my-sequence (sort my-sequence #'string<))
```

rather than just this:

```
(sort my-sequence #'string<)
```

Both these functions also take a keyword argument, `:key`, which, like the `:key` argument in other sequence functions, should be a function and will be used to extract the values to be passed to the sorting predicate in the place of the actual elements. The extracted keys are used only to determine the ordering of elements; the sequence returned will contain the actual elements of the argument sequence.

The **MERGE** function takes two sequences and a predicate and returns a sequence produced by merging the two sequences, according to the predicate. It's related to the two sorting functions in that if each sequence is already sorted by the same predicate, then the sequence returned by **MERGE** will also be sorted. Like the sorting functions, **MERGE** takes a `:key` argument. Like **CONCATENATE**, and for the same reason, the first argument to **MERGE** must be a type descriptor specifying the type of sequence to produce.

```
(merge 'vector #(1 3 5) #(2 4 6) #'<) ==> #(1 2 3 4 5 6)
(merge 'list #(1 3 5) #(2 4 6) #'<)   ==> (1 2 3 4 5 6)
```

Subsequence Manipulations

Another set of functions allows you to manipulate subsequences of existing sequences. The most basic of these is **SUBSEQ**, which extracts a subsequence starting at a particular index and continuing to a particular ending index or the end of the sequence. For instance:

```
(subseq "foobarbaz" 3)   ==> "barbaz"
(subseq "foobarbaz" 3 6) ==> "bar"
```

SUBSEQ is also **SETF**able, but it won't extend or shrink a sequence; if the new value and the subsequence to be replaced are different lengths, the shorter of the two determines how many characters are actually changed.

```
(defparameter *x* (copy-seq "foobarbaz"))

(setf (subseq *x* 3 6) "xxx")  ; subsequence and new value are same length
*x* ==> "fooxxxbaz"

(setf (subseq *x* 3 6) "abcd") ; new value too long, extra character ignored.
*x* ==> "fooabcbaz"

(setf (subseq *x* 3 6) "xx")   ; new value too short, only two characters changed
*x* ==> "fooxxcbaz"
```

You can use the **FILL** function to set multiple elements of a sequence to a single value. The required arguments are a sequence and the value with which to fill it. By default every element of the sequence is set to the value; `:start` and `:end` keyword arguments can limit the effects to a given subsequence.

If you need to find a subsequence within a sequence, the **SEARCH** function works like **POSITION** except the first argument is a sequence rather than a single item.

```
(position #\b "foobarbaz") ==> 3
(search "bar" "foobarbaz") ==> 3
```

On the other hand, to find where two sequences with a common prefix first diverge, you can use the **MISMATCH** function. It takes two sequences and returns the index of the first pair of mismatched elements.

```
(mismatch "foobarbaz" "foom") ==> 3
```

It returns **NIL** if the strings match. **MISMATCH** also takes many of the standard keyword arguments: a `:key` argument for specifying a function to use to extract the values to be compared; a `:test` argument to specify the comparison function; and `:start1`, `:end1`, `:start2`, and `:end2` arguments to specify subsequences within the two sequences. And a `:from-end` argument of **T** specifies the sequences should be searched in reverse order, causing **MISMATCH** to return the index, in the first sequence, where whatever common suffix the two sequences share begins.

```
(mismatch "foobar" "bar" :from-end t) ==> 3
```

Sequence Predicates

Four other handy functions are **EVERY**, **SOME**, **NOTANY**, and **NOTEVERY**, which iterate over sequences testing a boolean predicate. The first argument to all these functions is the predicate, and the remaining arguments are sequences. The predicate should take as many arguments as the number of sequences passed. The elements of the sequences are passed to the predicate--one element from each sequence--until one of the sequences runs out of elements or the overall termination test is met: **EVERY** terminates, returning false, as soon as the predicate fails. If the predicate is always satisfied, it returns true. **SOME** returns the first non-**NIL** value returned by the predicate or returns false if the predicate is never satisfied. **NOTANY** returns false as soon as the predicate is satisfied or true if it never is. And **NOTEVERY** returns true as soon as the predicate fails or false if the predicate is always satisfied. Here are some examples of testing just one sequence:

```
(every #'evenp #(1 2 3 4 5))    ==> NIL
(some #'evenp #(1 2 3 4 5))     ==> T
(notany #'evenp #(1 2 3 4 5))   ==> NIL
(notevery #'evenp #(1 2 3 4 5)) ==> T
```

These calls compare elements of two sequences pairwise:

```
(every #'> #(1 2 3 4) #(5 4 3 2))    ==> NIL
(some #'> #(1 2 3 4) #(5 4 3 2))     ==> T
(notany #'> #(1 2 3 4) #(5 4 3 2))   ==> NIL
(notevery #'> #(1 2 3 4) #(5 4 3 2)) ==> T
```

Sequence Mapping Functions

Finally, the last of the sequence functions are the generic mapping functions. **MAP**, like the sequence predicate functions, takes a *n*-argument function and *n* sequences. But instead of a boolean value, **MAP** returns a new sequence containing the result of applying the function to subsequent elements of the sequences. Like **CONCATENATE** and **MERGE**, **MAP** needs to be told what kind of sequence to create.

```
(map 'vector #'* #(1 2 3 4 5) #(10 9 8 7 6)) ==> #(10 18 24 28 30)
```

MAP-INTO is like **MAP** except instead of producing a new sequence of a given type, it places the results into a sequence passed as the first argument. This sequence can be the same as one of the sequences providing values for the function. For instance, to sum several vectors--`a`, `b`, and `c`--into one, you could write this:

```
(map-into a #'+ a b c)
```

If the sequences are different lengths, **MAP-INTO** affects only as many elements as are present in the shortest sequence, including the sequence being mapped into. However, if the sequence being mapped into is a vector with a fill pointer, the number of elements affected isn't limited by the fill pointer but rather by the actual size of the vector. After a call to **MAP-INTO**, the fill pointer will be set to the number of elements mapped. **MAP-INTO** won't, however, extend an adjustable vector.

The last sequence function is **REDUCE**, which does another kind of mapping: it maps over a single sequence, applying a two-argument function first to the first two elements of the sequence and then to

the value returned by the function and subsequent elements of the sequence. Thus, the following expression sums the numbers from one to ten:

```
(reduce #'+ #(1 2 3 4 5 6 7 8 9 10)) ==> 55
```

REDUCE is a surprisingly useful function--whenever you need to distill a sequence down to a single value, chances are you can write it with **REDUCE**, and it will often be quite a concise way to express what you want. For instance, to find the maximum value in a sequence of numbers, you can write `(reduce #'max numbers)`. **REDUCE** also takes a full complement of keyword arguments (`:key`, `:from-end`, `:start`, and `:end`) and one unique to **REDUCE** (`:initial-value`). The latter specifies a value that's logically placed before the first element of the sequence (or after the last if you also specify a true `:from-end` argument).

Hash Tables

The other general-purpose collection provided by Common Lisp is the hash table. Where vectors provide an integer-indexed data structure, hash tables allow you to use arbitrary objects as the indexes, or keys. When you add a value to a hash table, you store it under a particular key. Later you can use the same key to retrieve the value. Or you can associate a new value with the same key--each key maps to a single value.

With no arguments **MAKE-HASH-TABLE** makes a hash table that considers two keys equivalent if they're the same object according to **EQL**. This is a good default unless you want to use strings as keys, since two strings with the same contents aren't necessarily **EQL**. In that case you'll want a so-called **EQUAL** hash table, which you can get by passing the symbol **EQUAL** as the `:test` keyword argument to **MAKE-HASH-TABLE**. Two other possible values for the `:test` argument are the symbols **EQ** and **EQUALP**. These are, of course, the names of the standard object comparison functions, which I discussed in Chapter 4. However, unlike the `:test` argument passed to sequence functions, **MAKE-HASH-TABLE**'s `:test` can't be used to specify an arbitrary function--only the values **EQ**, **EQL**, **EQUAL**, and **EQUALP**. This is because hash tables actually need two functions, an equivalence function and a *hash* function that computes a numerical hash code from the key in a way compatible with how the equivalence function will ultimately compare two keys. However, although the language standard provides only for hash tables that use the standard

equivalence functions, most implementations provide some mechanism for defining custom hash tables.

The **GETHASH** function provides access to the elements of a hash table. It takes two arguments--a key and the hash table--and returns the value, if any, stored in the hash table under that key or **NIL**.[11] For example:

```
(defparameter *h* (make-hash-table))

(gethash 'foo *h*) ==> NIL

(setf (gethash 'foo *h*) 'quux)

(gethash 'foo *h*) ==> QUUX
```

Since **GETHASH** returns **NIL** if the key isn't present in the table, there's no way to tell from the return value the difference between a key not being in a hash table at all and being in the table with the value **NIL**. **GETHASH** solves this problem with a feature I haven't discussed yet--multiple return values. **GETHASH** actually returns two values; the primary value is the value stored under the given key or **NIL**. The secondary value is a boolean indicating whether the key is present in the hash table. Because of the way multiple values work, the extra return value is silently discarded unless the caller explicitly handles it with a form that can "see" multiple values.

I'll discuss multiple return values in greater detail in Chapter 20, but for now I'll give you a sneak preview of how to use the **MULTIPLE-VALUE-BIND** macro to take advantage of **GETHASH**'s extra return value. **MULTIPLE-VALUE-BIND** creates variable bindings like **LET** does, filling them with the multiple values returned by a form.

The following function shows how you might use **MULTIPLE-VALUE-BIND**; the variables it binds are `value` and `present`:

```
(defun show-value (key hash-table)
  (multiple-value-bind (value present) (gethash key hash-table)
    (if present
      (format nil "Value ~a actually present." value)
      (format nil "Value ~a because key not found." value))))

(setf (gethash 'bar *h*) nil) ; provide an explicit value of NIL

(show-value 'foo *h*) ==> "Value QUUX actually present."
(show-value 'bar *h*) ==> "Value NIL actually present."
(show-value 'baz *h*) ==> "Value NIL because key not found."
```

Since setting the value under a key to **NIL** leaves the key in the table, you'll need another function to completely remove a key/value pair. **REMHASH** takes the same arguments as **GETHASH** and removes the specified entry. You can also completely clear a hash table of all its key/value pairs with **CLRHASH**.

Hash Table Iteration

Common Lisp provides a couple ways to iterate over the entries in a hash table. The simplest of these is via the function **MAPHASH**. Analogous to the **MAP** function, **MAPHASH** takes a two-argument function and a hash table and invokes the function once for each key/value pair in the hash table. For instance, to print all the key/value pairs in a hash table, you could use **MAPHASH** like this:

```
(maphash #'(lambda (k v) (format t "~a => ~a~%" k v)) *h*)
```

The consequences of adding or removing elements from a hash table while iterating over it aren't specified (and are likely to be bad) with two exceptions: you can use **SETF** with **GETHASH** to change the value of the current entry, and you can use **REMHASH** to remove the current entry. For instance, to remove all the entries whose value is less than ten, you could write this:

```
(maphash #'(lambda (k v) (when (< v 10) (remhash k *h*))) *h*)
```

The other way to iterate over a hash table is with the extended **LOOP** macro, which I'll discuss in Chapter 22.[12] The **LOOP** equivalent of the first **MAPHASH** expression would look like this:

```
(loop for k being the hash-keys in *h* using (hash-value v)
  do (format t "~a => ~a~%" k v))
```

I could say a lot more about the nonlist collections supported by Common Lisp. For instance, I haven't discussed multidimensional arrays at all or the library of functions for manipulating bit arrays. However, what I've covered in this chapter should suffice for most of

your general-purpose programming needs. Now it's finally time to look at Lisp's eponymous data structure: lists.

---

[1]Once you're familiar with all the data types Common Lisp offers, you'll also see that lists can be useful for prototyping data structures that will later be replaced with something more efficient once it becomes clear how exactly the data is to be used.

[2]Vectors are called *vectors*, not *arrays* as their analogs in other languages are, because Common Lisp supports true multidimensional arrays. It's equally correct, though more cumbersome, to refer to them as *one-dimensional arrays*.

[3]Array elements "must" be set before they're accessed in the sense that the behavior is undefined; Lisp won't necessarily stop you.

[4]While frequently used together, the `:fill-pointer` and `:adjustable` arguments are independent--you can make an adjustable array without a fill pointer. However, you can use **VECTOR-PUSH** and **VECTOR-POP** only with vectors that have a fill pointer and **VECTOR-PUSH-EXTEND** only with vectors that have a fill pointer and are adjustable. You can also use the function **ADJUST-ARRAY** to modify adjustable arrays in a variety of ways beyond just extending the length of a vector.

[5]Another parameter, `:test-not` parameter, specifies a two-argument predicate to be used like a `:test` argument except with the boolean result logically reversed. This parameter is deprecated, however, in preference for using the **COMPLEMENT** function. **COMPLEMENT** takes a function argu-ment and returns a function that takes the same number of arguments as the original and returns the logical complement of the original function. Thus, you can, and should, write this:

```
(count x sequence :test (complement #'some-test))
```

rather than the following:

```
(count x sequence :test-not #'some-test)
```

[6]Note, however, that the effect of `:start` and `:end` on **REMOVE** and **SUBSTITUTE** is only to limit the elements they consider for removal or substitution; elements before `:start` and after `:end` will be passed through untouched.

[7]This same functionality goes by the name `grep` in Perl and `filter` in Python.

[8]The difference between the predicates passed as `:test` arguments and as the function arguments to the `-IF` and `-IF-NOT` functions is that the `:test` predicates are two-argument predicates used to compare the elements of the sequence to the specific item while the `-IF` and `-IF-NOT` predicates are one-argument functions that simply test the individual elements of the sequence. If the vanilla variants didn't exist, you could implement them in terms of the -IF versions by embedding a specific item in the test function.

```
(count char string) ===
  (count-if #'(lambda (c) (eql char c)) string)

(count char string :test #'CHAR-EQUAL) ===
  (count-if #'(lambda (c) (char-equal char c)) string)
```

[9]If you tell **CONCATENATE** to return a specialized vector, such as a string, all the elements of the argument sequences must be instances of the vector's element type.

[10]When the sequence passed to the sorting functions is a vector, the "destruction" is actually guaranteed to entail permuting the elements in place, so you could get away without saving the returned value. However, it's good style to always do something with the return value since the sorting functions can modify lists in much more arbitrary ways.

[11]By an accident of history, the order of arguments to **GETHASH** is the opposite of **ELT--ELT** takes the collection first and then the index while **GETHASH** takes the key first and then the collection.

[12]**LOOP**'s hash table iteration is typically implemented on top of a more primitive form, **WITH-HASH-TABLE-ITERATOR**, that you don't need to worry about; it was added to the language specifically to support implementing things such as **LOOP** and is of little use unless you need to write completely new control constructs for iterating over hash tables.

Copyright © 2003-2005, Peter Seibel

12. They Called It LISP for a Reason: List Processing

Lists play an important role in Lisp--for reasons both historical and practical. Historically, lists were Lisp's original composite data type, though it has been decades since they were its *only* such data type. These days, a Common Lisp programmer is as likely to use a vector, a hash table, or a user-defined class or structure as to use a list.

Practically speaking, lists remain in the language because they're an excellent solution to certain problems. One such problem--how to represent code as data in order to support code-transforming and code-generating macros--is particular to Lisp, which may explain why other languages don't feel the lack of Lisp-style lists. More generally, lists are an excellent data structure for representing any kind of heterogeneous and/or hierarchical data. They're also quite lightweight and support a functional style of programming that's another important part of Lisp's heritage.

Thus, you need to understand lists on their own terms; as you gain a better understanding of how lists work, you'll be in a better position to appreciate when you should and shouldn't use them.

"There Is No List"

> **Spoon Boy**: Do not try and bend the list. That's impossible. Instead . . . only try to realize the truth.
>
> **Neo**: What truth?
>
> **Spoon Boy**: There is no list.
>
> **Neo**: There is no list?
>
> **Spoon Boy**: Then you'll see that it is not the list that bends; it is only yourself.[1]

The key to understanding lists is to understand that they're largely an illusion built on top of objects that are instances of a more primitive data type. Those simpler objects are pairs of values called *cons cells*, after the function `CONS` used to create them.

`CONS` takes two arguments and returns a new cons cell containing the two values.[2] These values can be references to any kind of object. Unless the second value is `NIL` or another cons cell, a cons is printed as the two values in parentheses separated by a dot, a so-called dotted pair.

```
(cons 1 2) ==> (1 . 2)
```

The two values in a cons cell are called the `CAR` and the `CDR` after the names of the functions used to access them. At the dawn of time, these names were mnemonic, at least to the folks implementing the first Lisp on an IBM 704. But even then they were just lifted from the assembly mnemonics used to implement the operations. However, it's not all bad that these names are somewhat meaningless--when

considering individual cons cells, it's best to think of them simply as an arbitrary pair of values without any particular semantics. Thus:

```
(car (cons 1 2)) ==> 1
(cdr (cons 1 2)) ==> 2
```

Both CAR and CDR are also SETFable places--given an existing cons cell, it's possible to assign a new value to either of its values.[3]

```
(defparameter *cons* (cons 1 2))
*cons*                  ==> (1 . 2)
(setf (car *cons*) 10) ==> 10
*cons*                  ==> (10 . 2)
(setf (cdr *cons*) 20) ==> 20
*cons*                  ==> (10 . 20)
```

Because the values in a cons cell can be references to any kind of object, you can build larger structures out of cons cells by linking them together. Lists are built by linking together cons cells in a chain. The elements of the list are held in the CARs of the cons cells while the links to subsequent cons cells are held in the CDRs. The last cell in the chain has a CDR of NIL, which--as I mentioned in Chapter 4--represents the empty list as well as the boolean value false.

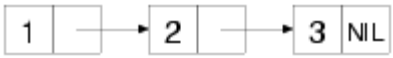
This arrangement is by no means unique to Lisp; it's called a *singly linked list*. However, few languages outside the Lisp family provide such extensive support for this humble data type.

So when I say a particular value is a list, what I really mean is it's either NIL or a reference to a cons cell. The CAR of the cons cell is the first item of the list, and the CDR is a reference to another list, that is, another cons cell or NIL, containing the remaining elements. The Lisp printer understands this convention and prints such chains of cons cells as parenthesized lists rather than as dotted pairs.

```
(cons 1 nil)                    ==> (1)
(cons 1 (cons 2 nil))           ==> (1 2)
(cons 1 (cons 2 (cons 3 nil))) ==> (1 2 3)
```

When talking about structures built out of cons cells, a few diagrams can be a big help. Box-and-arrow diagrams represent cons cells as a pair of boxes like this:

The box on the left represents the **CAR**, and the box on the right is the **CDR**. The values stored in a particular cons cell are either drawn in the appropriate box or represented by an arrow from the box to a representation of the referenced value.[4] For instance, the list `(1 2 3)`, which consists of three cons cells linked together by their **CDR**s, would be diagrammed like this:



However, most of the time you work with lists you won't have to deal with individual cons cells--the functions that create and manipulate lists take care of that for you. For example, the **LIST** function builds a cons cells under the covers for you and links them together; the following **LIST** expressions are equivalent to the previous **CONS** expressions:

```
(list 1)     ==> (1)
(list 1 2)   ==> (1 2)
(list 1 2 3) ==> (1 2 3)
```

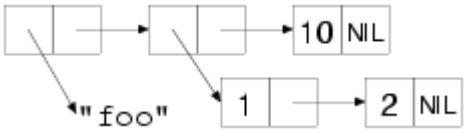
Similarly, when you're thinking in terms of lists, you don't have to use the meaningless names **CAR** and **CDR**; **FIRST** and **REST** are synonyms for **CAR** and **CDR** that you should use when you're dealing with cons cells as lists.

```
(defparameter *list* (list 1 2 3 4))
(first *list*)        ==> 1
(rest *list*)         ==> (2 3 4)
(first (rest *list*)) ==> 2
```

Because cons cells can hold any kind of values, so can lists. And a single list can hold objects of different types.

```
(list "foo" (list 1 2) 10) ==> ("foo" (1 2) 10)
```

The structure of that list would look like this:



Because lists can have other lists as elements, you can also use them to represent trees of arbitrary depth and complexity. As such, they make excellent representations for any heterogeneous, hierarchical data. Lisp-based XML processors, for instance, usually represent XML documents internally as lists. Another obvious example of tree-structured data is Lisp code itself. In Chapters 30 and 31 you'll write an HTML generation library that uses lists of lists to represent the HTML to be generated. I'll talk more next chapter about using cons cells to represent other data structures.

Common Lisp provides quite a large library of functions for manipulating lists. In the sections "List-Manipulation Functions" and "Mapping," you'll look at some of the more important of these functions. However, they will be easier to understand in the context of a few ideas borrowed from functional programming.

Functional Programming and Lists

The essence of functional programming is that programs are built entirely of functions with no side effects that compute their results based solely on the values of their arguments. The advantage of the functional style is that it makes programs easier to understand. Eliminating side effects eliminates almost all possibilities for action at a distance. And since the result of a function is determined only by the values of its arguments, its behavior is easier to understand and test. For instance, when you see an expression such as `(+ 3 4)`, you know the result is uniquely determined by the definition of the `+` function and the values `3` and `4`. You don't have to worry about what may have happened earlier in the execution of the program since there's nothing that can change the result of evaluating that expression.

Functions that deal with numbers are naturally functional since numbers are immutable. A list, on the other hand, can be mutated, as you've just seen, by **SETF**ing the **CAR**s and **CDR**s of the cons cells that make up its backbone. However, lists can be treated as a functional data type if you consider their value to be determined by the elements they contain. Thus, any list of the form `(1 2 3 4)` is functionally equivalent to any other list containing those four values, regardless of what cons cells are actually used to represent the list. And any function that takes a list as an argument and returns a value based solely on the contents of the list can likewise be considered functional. For instance, the **REVERSE** sequence function, given the list `(1 2 3 4)`, always returns a list `(4 3 2 1)`. Different calls to **REVERSE**

with functionally equivalent lists as the argument will return functionally equivalent result lists. Another aspect of functional programming, which I'll discuss in the section "Mapping," is the use of higher-order functions: functions that treat other functions as data, taking them as arguments or returning them as results.

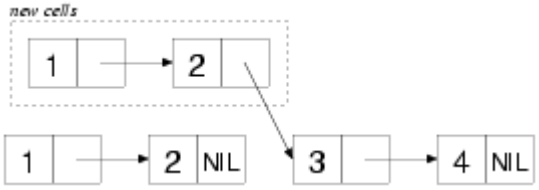
Most of Common Lisp's list-manipulation functions are written in a functional style. I'll discuss later how to mix functional and other coding styles, but first you should understand a few subtleties of the functional style as applied to lists.

The reason most list functions are written functionally is it allows them to return results that share cons cells with their arguments. To take a concrete example, the function **APPEND** takes any number of list arguments and returns a new list containing the elements of all its arguments. For instance:

```
(append (list 1 2) (list 3 4)) ==> (1 2 3 4)
```

From a functional point of view, **APPEND**'s job is to return the list `(1 2 3 4)` without modifying any of the cons cells in the lists `(1 2)` and `(3 4)`. One obvious way to achieve that goal is to create a completely new list consisting of four new cons cells. However, that's more work than is necessary. Instead, **APPEND** actually makes only two new cons cells to hold the values `1` and `2`, linking them together and pointing the **CDR** of the second cons cell at the head of the last argument, the list `(3 4)`. It then returns the cons cell containing the `1`. None of the original cons cells has been modified, and the result is indeed the list `(1 2 3 4)`. The only wrinkle is that the list returned by

APPEND shares some cons cells with the list `(3 4)`. The resulting structure looks like this:



In general, **APPEND** must copy all but its last argument, but it can always return a result that *shares structure* with the last argument.

Other functions take similar advantage of lists' ability to share structure. Some, like **APPEND**, are specified to always return results that share structure in a particular way. Others are simply allowed to return shared structure at the discretion of the implementation.

"Destructive" Operations

If Common Lisp were a purely functional language, that would be the end of the story. However, because it's possible to modify a cons cell after it has been created by **SETF**ing its **CAR** or **CDR**, you need to think a bit about how side effects and structure sharing mix.

Because of Lisp's functional heritage, operations that modify existing objects are called *destructive*--in functional programming, changing an object's state "destroys" it since it no longer represents the same value. However, using the same term to describe all state-modifying operations leads to a certain amount of confusion since there are two very different kinds of destructive operations, *for-side-effect* operations and *recycling* operations.[5]

For-side-effect operations are those used specifically for their side effects. All uses of **SETF** are destructive in this sense, as are functions that use **SETF** under the covers to change the state of an existing object such as **VECTOR-PUSH** or **VECTOR-POP**. But it's a bit unfair to describe these operations as destructive--they're not intended to be used in code written in a functional style, so they shouldn't be described using functional terminology. However, if you mix nonfunctional, for-side-effect operations with functions that return structure-sharing results, then you need to be careful not to inadvertently modify the shared structure. For instance, consider these three definitions:

```
(defparameter *list-1* (list 1 2))
(defparameter *list-2* (list 3 4))
(defparameter *list-3* (append *list-1* *list-2*))
```

After evaluating these forms, you have three lists, but `*list-3*` and `*list-2*` share structure just like the lists in the previous diagram.

```
*list-1*                  ==> (1 2)
*list-2*                  ==> (3 4)
*list-3*                  ==> (1 2 3 4)
```

Now consider what happens when you modify `*list-2*`.

```
(setf (first *list-2*) 0) ==> 0
*list-2*                  ==> (0 4)     ; as expected
*list-3*                  ==> (1 2 0 4) ; maybe not what you wanted
```

The change to `*list-2*` also changes `*list-3*` because of the shared structure: the first cons cell in `*list-2*` is also the third cons cell in `*list-3*`. **SETF**ing the **FIRST** of `*list-2*` changes the value in the **CAR** of that cons cell, affecting both lists.

On the other hand, the other kind of destructive operations, recycling operations, *are* intended to be used in functional code. They use side effects only as an optimization. In particular, they reuse certain cons cells from their arguments when building their result. However, unlike functions such as **APPEND** that reuse cons cells by including them, unmodified, in the list they return, recycling functions reuse cons cells as raw material, modifying the **CAR** and **CDR** as necessary to build the desired result. Thus, recycling functions can be used safely only when the original lists aren't going to be needed after the call to the recycling function.

To see how a recycling function works, let's compare **REVERSE**, the nondestructive function that returns a reversed version of a sequence, to **NREVERSE**, a recycling version of the same function. Because **REVERSE** doesn't modify its argument, it must allocate a new

cons cell for each element in the list being reversed. But suppose you write something like this:

```
(setf *list* (reverse *list*))
```

By assigning the result of **REVERSE** back to `*list*`, you've removed the reference to the original value of `*list*`. Assuming the cons cells in the original list aren't referenced anywhere else, they're now eligible to be garbage collected. However, in many Lisp implementations it'd be more efficient to immediately reuse the existing cons cells rather than allocating new ones and letting the old ones become garbage.

NREVERSE allows you to do exactly that. The *N* stands for *non-consing*, meaning it doesn't need to allocate any new cons cells. The exact side effects of **NREVERSE** are intentionally not specified--it's allowed to modify any **CAR** or **CDR** of any cons cell in the list--but a typical implementation might walk down the list changing the **CDR** of each cons cell to point to the previous cons cell, eventually returning the cons cell that was previously the last cons cell in the old list and is now the head of the reversed list. No new cons cells need to be allocated, and no garbage is created.

Most recycling functions, like **NREVERSE**, have nondestructive counterparts that compute the same result. In general, the recycling functions have names that are the same as their non-destructive counterparts except with a leading *N*. However, not all do, including several of the more commonly used recycling functions such as **NCONC**, the recycling version of **APPEND**, and **DELETE**, **DELETE-IF**,

DELETE-IF-NOT, and DELETE-DUPLICATES, the recycling versions of the REMOVE family of sequence functions.

In general, you use recycling functions in the same way you use their nondestructive counterparts except it's safe to use them only when you know the arguments aren't going to be used after the function returns. The side effects of most recycling functions aren't specified tightly enough to be relied upon.

However, the waters are further muddied by a handful of recycling functions with specified side effects that *can* be relied upon. They are NCONC, the recycling version of APPEND, and NSUBSTITUTE and its -IF and -IF-NOT variants, the recycling versions of the sequence functions SUBSTITUTE and friends.

Like APPEND, NCONC returns a concatenation of its list arguments, but it builds its result in the following way: for each nonempty list it's passed, NCONC sets the CDR of the list's last cons cell to point to the first cons cell of the next nonempty list. It then returns the first list, which is now the head of the spliced-together result. Thus:

```
(defparameter *x* (list 1 2 3))

(nconc *x* (list 4 5 6)) ==> (1 2 3 4 5 6)

*x* ==> (1 2 3 4 5 6)
```

NSUBSTITUTE and variants can be relied on to walk down the list structure of the list argument and to SETF the CARs of any cons cells holding the old value to the new value and to otherwise leave the list intact. It then returns the original list, which now has the same value as would've been computed by SUBSTITUTE. [6]

The key thing to remember about `NCONC` and `NSUBSTITUTE` is that they're the exceptions to the rule that you can't rely on the side effects of recycling functions. It's perfectly acceptable--and arguably good style--to ignore the reliability of their side effects and use them, like any other recycling function, only for the value they return.

Combining Recycling with Shared Structure

Although you can use recycling functions whenever the arguments to the recycling function won't be used after the function call, it's worth noting that each recycling function is a loaded gun pointed footward: if you accidentally use a recycling function on an argument that *is* used later, you're liable to lose some toes.

To make matters worse, shared structure and recycling functions tend to work at cross-purposes. Nondestructive list functions return lists that share structure under the assumption that cons cells are never modified, but recycling functions work by violating that assumption. Or, put another way, sharing structure is based on the premise that you don't care exactly what cons cells make up a list while using recycling functions requires that you know exactly what cons cells are referenced from where.

In practice, recycling functions tend to be used in a few idiomatic ways. By far the most common recycling idiom is to build up a list to be returned from a function by "consing" onto the front of a list, usually by `PUSH`ing elements onto a list stored in a local variable and then returning the result of `NREVERSE`ing it.[7]

This is an efficient way to build a list because each `PUSH` has to create only one cons cell and modify a local variable and the `NREVERSE` just has to zip down the list reassigning the `CDR`s. Because the list is created entirely within the function, there's no danger any code outside the function has a reference to any of its cons cells.

Here's a function that uses this idiom to build a list of the first *n* numbers, starting at zero:[8]

```
(defun upto (max)
  (let ((result nil))
    (dotimes (i max)
      (push i result))
    (nreverse result)))

(upto 10) ==> (0 1 2 3 4 5 6 7 8 9)
```

The next most common recycling idiom[9] is to immediately reassign the value returned by the recycling function back to the place containing the potentially recycled value. For instance, you'll often see expressions like the following, using **DELETE**, the recycling version of **REMOVE**:

```
(setf foo (delete nil foo))
```

This sets the value of `foo` to its old value except with all the **NIL**s removed. However, even this idiom must be used with some care--if `foo` shares structure with lists referenced elsewhere, using **DELETE** instead of **REMOVE** can destroy the structure of those other lists. For example, consider the two lists `*list-2*` and `*list-3*` from earlier that share their last two cons cells.

```
*list-2* ==> (0 4)
*list-3* ==> (1 2 0 4)
```

You can delete `4` from `*list-3*` like this:

```
(setf *list-3* (delete 4 *list-3*)) ==> (1 2 0)
```

However, **DELETE** will likely perform the necessary deletion by setting the **CDR** of the third cons cell to **NIL**, disconnecting the fourth cons cell, the one holding the `4`, from the list. Because the third cons cell

of `*list-3*` is also the first cons cell in `*list-2*`, the following modifies `*list-2*` as well:

```
*list-2* ==> (0)
```

If you had used **REMOVE** instead of **DELETE**, it would've built a list containing the values `1`, `2`, and `0`, creating new cons cells as necessary rather than modifying any of the cons cells in `*list-3*`. In that case, `*list-2*` wouldn't have been affected.

The **PUSH**/**NREVERSE** and **SETF**/**DELETE** idioms probably account for 80 percent of the uses of recycling functions. Other uses are possible but require keeping careful track of which functions return shared structure and which do not.

In general, when manipulating lists, it's best to write your own code in a functional style--your functions should depend only on the contents of their list arguments and shouldn't modify them. Following that rule will, of course, rule out using any destructive functions, recycling or otherwise. Once you have your code working, if profiling shows you need to optimize, you can replace nondestructive list operations with their recycling counterparts but only if you're certain the argument lists aren't referenced from anywhere else.

One last gotcha to watch out for is that the sorting functions **SORT**, **STABLE-SORT**, and **MERGE** mentioned in Chapter 11 are also recycling functions when applied to lists.[10] However, these functions don't have nondestructive counterparts, so if you need to sort a list without destroying it, you need to pass the sorting function a copy made with **COPY-LIST**. In either case you need to be sure to save the result of

the sorting function because the original argument is likely to be in tatters. For instance:

```
CL-USER> (defparameter *list* (list 4 3 2 1))
*LIST*
CL-USER> (sort *list* #'<)
(1 2 3 4)                       ; looks good
CL-USER> *list*
(4)                             ; whoops!
```

List-Manipulation Functions

With that background out of the way, you're ready to look at the library of functions Common Lisp provides for manipulating lists.

You've already seen the basic functions for getting at the elements of a list: **FIRST** and **REST**. Although you can get at any element of a list by combining enough calls to **REST** (to move down the list) with a **FIRST** (to extract the element), that can be a bit tedious. So Common Lisp provides functions named for the other ordinals from **SECOND** to **TENTH** that return the appropriate element. More generally, the function **NTH** takes two arguments, an index and a list, and returns the *n*th (zero-based) element of the list. Similarly, **NTHCDR** takes an index and a list and returns the result of calling **CDR** *n* times. (Thus, `(nthcdr 0 ...)` simply returns the original list, and `(nthcdr 1 ...)` is equivalent to **REST**.) Note, however, that none of these functions is any more efficient, in terms of work done by the computer, than the equivalent combinations of **FIRST**s and **REST**s--there's no way to get to the *n*th element of a list without following *n* **CDR** references.[11]

The 28 composite **CAR**/**CDR** functions are another family of functions you may see used from time to time. Each function is named by placing a sequence of up to four `A`s and `D`s between a `C` and `R`, with each `A` representing a call to **CAR** and each `D` a call to **CDR**. Thus:

```
(caar list) === (car (car list))
(cadr list) === (car (cdr list))
(cadadr list) === (car (cdr (car (cdr list))))
```

Note, however, that many of these functions make sense only when applied to lists that contain other lists. For instance, **CAAR** extracts the

CAR of the **CAR** of the list it's given; thus, the list it's passed must contain another list as its first element. In other words, these are really functions on trees rather than lists:

```
(caar (list 1 2 3))                  ==> error
(caar (list (list 1 2) 3))           ==> 1
(cadr (list (list 1 2) (list 3 4)))  ==> (3 4)
(caadr (list (list 1 2) (list 3 4))) ==> 3
```

These functions aren't used as often now as in the old days. And even the most die-hard old-school Lisp hackers tend to avoid the longer combinations. However, they're used quite a bit in older Lisp code, so it's worth at least understanding how they work.[12]

The **FIRST-TENTH** and **CAR**, **CADR**, and so on, functions can also be used as **SETF**able places if you're using lists nonfunctionally.

Table 12-1 summarizes some other list functions that I won't cover in detail.

Table 12-1. Other List Functions

Function	Description
LAST	Returns the last cons cell in a list. With an integer, argument returns the last *n* cons cells.
BUTLAST	Returns a copy of the list, excluding the last cons cell. With an integer argument, excludes the last *n* cells.
NBUTLAST	The recycling version of **BUTLAST**; may modify and return the argument list but has no reliable side effects.
LDIFF	Returns a copy of a list up to a given cons cell.
TAILP	Returns true if a given object is a cons cell that's part of the structure of a list.
LIST*	Builds a list to hold all but the last of its arguments and then makes the last argument the **CDR** of the last cell in the list. In other words, a cross between **LIST** and **APPEND**.
MAKE-LIST	Builds an *n* item list. The initial elements of the list are **NIL** or the value specified with the `:initial-element` keyword argument.

REVAPPEND	Combination of **REVERSE** and **APPEND**; reverses first argument as with **REVERSE** and then appends the second argument.
NRECONC	Recycling version of **REVAPPEND**; reverses first argument as if by **NREVERSE** and then appends the second argument. No reliable side effects.
CONSP	Predicate to test whether an object is a cons cell.
ATOM	Predicate to test whether an object is *not* a cons cell.
LISTP	Predicate to test whether an object is either a cons cell or **NIL**.
NULL	Predicate to test whether an object is **NIL**. Functionally equivalent to **NOT** but stylistically preferable when testing for an empty list as opposed to boolean false.

Mapping

Another important aspect of the functional style is the use of higher-order functions, functions that take other functions as arguments or return functions as values. You saw several examples of higher-order functions, such as **MAP**, in the previous chapter. Although **MAP** can be used with both lists and vectors (that is, with any kind of sequence), Common Lisp also provides six mapping functions specifically for lists. The differences between the six functions have to do with how they build up their result and whether they apply the function to the elements of the list or to the cons cells of the list structure.

MAPCAR is the function most like **MAP**. Because it always returns a list, it doesn't require the result-type argument **MAP** does. Instead, its first argument is the function to apply, and subsequent arguments are the lists whose elements will provide the arguments to the function. Otherwise, it behaves like **MAP**: the function is applied to successive elements of the list arguments, taking one element from each list per application of the function. The results of each function call are collected into a new list. For example:

```
(mapcar #'(lambda (x) (* 2 x)) (list 1 2 3)) ==> (2 4 6)
(mapcar #'+ (list 1 2 3) (list 10 20 30)) ==> (11 22 33)
```

MAPLIST is just like **MAPCAR** except instead of passing the elements of the list to the function, it passes the actual cons cells.[13] Thus, the function has access not only to the value of each element of the list (via the **CAR** of the cons cell) but also to the rest of the list (via the **CDR**).

MAPCAN and MAPCON work like MAPCAR and MAPLIST except for the way they build up their result. While MAPCAR and MAPLIST build a completely new list to hold the results of the function calls, MAPCAN and MAPCON build their result by splicing together the results--which must be lists--as if by NCONC. Thus, each function invocation can provide any number of elements to be included in the result.[14] MAPCAN, like MAPCAR, passes the elements of the list to the mapped function while MAPCON, like MAPLIST, passes the cons cells.

Finally, the functions MAPC and MAPL are control constructs disguised as functions--they simply return their first list argument, so they're useful only when the side effects of the mapped function do something interesting. MAPC is the cousin of MAPCAR and MAPCAN while MAPL is in the MAPLIST/MAPCON family.

Other Structures

While cons cells and lists are typically considered to be synonymous, that's not quite right--as I mentioned earlier, you can use lists of lists to represent trees. Just as the functions discussed in this chapter allow you to treat structures built out of cons cells as lists, other functions allow you to use cons cells to represent trees, sets, and two kinds of key/value maps. I'll discuss some of those functions in the next chapter.

---

[1]Adapted from *The Matrix* (`http://us.imdb.com/Quotes?0133093`)

[2]**CONS** was originally short for the verb *construct*.

[3]When the place given to **SETF** is a **CAR** or **CDR**, it expands into a call to the function **RPLACA** or **RPLACD**; some old-school Lispers--the same ones who still use **SETQ**--will still use **RPLACA** and **RPLACD** directly, but modern style is to use **SETF** of **CAR** or **CDR**.

[4]Typically, simple objects such as numbers are drawn within the appropriate box, and more complex objects will be drawn outside the box with an arrow from the box indicating the reference. This actually corresponds well with how many Common Lisp implementations work--although all objects are conceptually stored by reference, certain simple immutable objects can be stored directly in a cons cell.

[5]The phrase *for-side-effect* is used in the language standard, but *recycling* is my own invention; most Lisp literature simply uses the term *destructive* for both kinds of operations, leading to the confusion I'm trying to dispel.

[6]The string functions **NSTRING-CAPITALIZE**, **NSTRING-DOWNCASE**, and **NSTRING-UPCASE** are similar--they return the same results as their N-less counterparts but are specified to modify their string argument in place.

[7]For example, in an examination of all uses of recycling functions in the Common Lisp Open Code Collection (CLOCC), a diverse set of libraries written by various authors,

instances of the **PUSH**/**NREVERSE** idiom accounted for nearly half of all uses of recycling functions.

[8]There are, of course, other ways to do this same thing. The extended **LOOP** macro, for instance, makes it particularly easy and likely generates code that's even more efficient than the **PUSH**/ **NREVERSE** version.

[9]This idiom accounts for 30 percent of uses of recycling in the CLOCC code base.

[10]**SORT** and **STABLE-SORT** can be used as for-side-effect operations on vectors, but since they still return the sorted vector, you should ignore that fact and use them for return values for the sake of consistency.

[11]**NTH** is roughly equivalent to the sequence function **ELT** but works only with lists. Also, confusingly, **NTH** takes the index as the first argument, the opposite of **ELT**. Another difference is that **ELT** will signal an error if you try to access an element at an index greater than or equal to the length of the list, but **NTH** will return **NIL**.

[12]In particular, they used to be used to extract the various parts of expressions passed to macros before the invention of destructuring parameter lists. For example, you could take apart the following expression:

```
(when (> x 10) (print x))
```

Like this:

```
;; the condition
(cadr '(when (> x 10) (print x))) ==> (> X 10)

;; the body, as a list
(cddr '(when (> x 10) (print x))) ==> ((PRINT X))
```

[13]Thus, **MAPLIST** is the more primitive of the two functions--if you had only **MAPLIST**, you could build **MAPCAR** on top of it, but you couldn't build **MAPLIST** on top of **MAPCAR**.

[14]In Lisp dialects that didn't have filtering functions like **REMOVE**, the idiomatic way to filter a list was with **MAPCAN**.

```
(mapcan #'(lambda (x) (if (= x 10) nil (list x)))  list) === (remove 10 list)
```

Copyright © 2003-2005, Peter Seibel

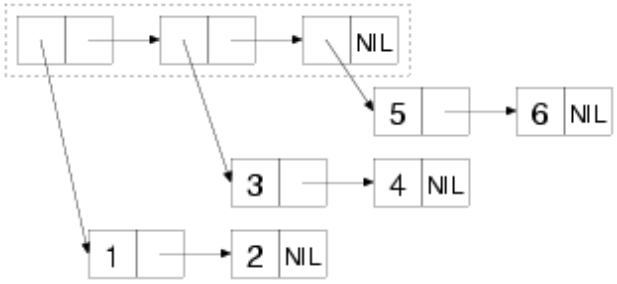
13. Beyond Lists: Other Uses for Cons Cells

As you saw in the previous chapter, the list data type is an illusion created by a set of functions that manipulate cons cells. Common Lisp also provides functions that let you treat data structures built out of cons cells as trees, sets, and lookup tables. In this chapter I'll give you a quick tour of some of these other data structures and the functions for manipulating them. As with the list-manipulation functions, many of these functions will be useful when you start writing more complicated macros and need to manipulate Lisp code as data.

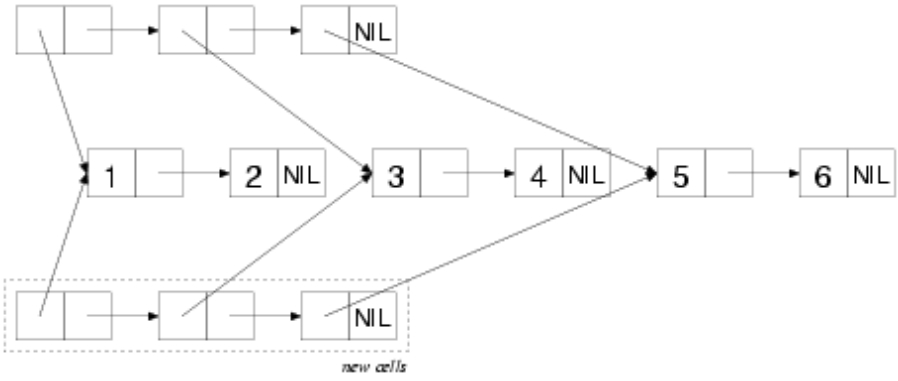
Trees

Treating structures built from cons cells as trees is just about as natural as treating them as lists. What is a list of lists, after all, but another way of thinking of a tree? The difference between a function that treats a bunch of cons cells as a list and a function that treats the same bunch of cons cells as a tree has to do with which cons cells the functions traverse to find the values of the list or tree. The cons cells traversed by a list function, called the *list structure*, are found by starting at the first cons cell and following `CDR` references until reaching a `NIL`. The elements of the list are the objects referenced by the `CAR`s of the cons cells in the list structure. If a cons cell in the list structure has a `CAR` that references another cons cell, the referenced cons cell is considered to be the head of a list that's an element of the outer list.[1] *Tree structure*, on the other hand, is traversed by following both `CAR` and `CDR` references for as long as they point to other cons cells. The values in a tree are thus the atomic--non-cons-cell-values referenced by either the `CAR`s or the `CDR`s of the cons cells in the tree structure.

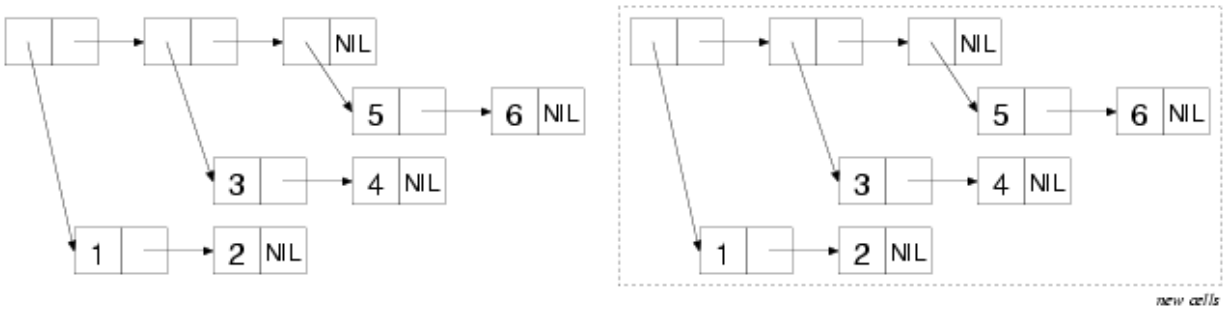
For instance, the following box-and-arrow diagram shows the cons cells that make up the list of lists: `((1 2) (3 4) (5 6))`. The list structure includes only the three cons cells inside the dashed box while the tree structure includes all the cons cells.



To see the difference between a list function and a tree function, you can consider how the functions `COPY-LIST` and `COPY-TREE` will copy this bunch of cons cells. `COPY-LIST`, as a list function, copies the cons cells that make up the list structure. That is, it makes a new cons cell corresponding to each of the cons cells inside the dashed box. The `CAR`s of each of these new cons cells reference the same object as the `CAR`s of the original cons cells in the list structure. Thus, `COPY-LIST` doesn't copy the sublists `(1 2)`, `(3 4)`, or `(5 6)`, as shown in this diagram:



`COPY-TREE`, on the other hand, makes a new cons cell for each of the cons cells in the diagram and links them together in the same structure, as shown in this diagram:



Where a cons cell in the original referenced an atomic value, the corresponding cons cell in the copy will reference the same value. Thus, the only objects referenced in common by the original tree and the copy produced by **COPY-TREE** are the numbers 5, 6, and the symbol **NIL**.

Another function that walks both the **CAR**s and the **CDR**s of a tree of cons cells is **TREE-EQUAL**, which compares two trees, considering them equal if the tree structure is the same shape and if the leaves are **EQL** (or if they satisfy the test supplied with the `:test` keyword argument).

Some other tree-centric functions are the tree analogs to the **SUBSTITUTE** and **NSUBSTITUTE** sequence functions and their `-IF` and `-IF-NOT` variants. The function **SUBST**, like **SUBSTITUTE**, takes a new item, an old item, and a tree (as opposed to a sequence), along with `:key` and `:test` keyword arguments, and it returns a new tree with the same shape as the original tree but with all instances of the old item replaced with the new item. For example:

```
CL-USER> (subst 10 1 '(1 2 (3 2 1) ((1 1) (2 2))))
(10 2 (3 2 10) ((10 10) (2 2)))
```

SUBST-IF is analogous to **SUBSTITUTE-IF**. Instead of an old item, it takes a one-argument function--the function is called with each atomic value in the tree, and whenever it returns true, the position in

the new tree is filled with the new value. SUBST-IF-NOT is the same except the values where the test returns NIL are replaced. NSUBST, NSUBST-IF, and NSUBST-IF-NOT are the recycling versions of the SUBST functions. As with most other recycling functions, you should use these functions only as drop-in replacements for their nondestructive counterparts in situations where you know there's no danger of modifying a shared structure. In particular, you must continue to save the return value of these functions since you have no guarantee that the result will be EQ to the original tree.[2]

Sets

Sets can also be implemented in terms of cons cells. In fact, you can treat any list as a set--Common Lisp provides several functions for performing set-theoretic operations on lists. However, you should bear in mind that because of the way lists are structured, these operations get less and less efficient the bigger the sets get.

That said, using the built-in set functions makes it easy to write set-manipulation code. And for small sets they may well be more efficient than the alternatives. If profiling shows you that these functions are a performance bottleneck in your code, you can always replace the lists with sets built on top of hash tables or bit vectors.

To build up a set, you can use the function **ADJOIN**. **ADJOIN** takes an item and a list representing a set and returns a list representing the set containing the item and all the items in the original set. To determine whether the item is present, it must scan the list; if the item isn't found, **ADJOIN** creates a new cons cell holding the item and pointing to the original list and returns it. Otherwise, it returns the original list.

ADJOIN also takes `:key` and `:test` keyword arguments, which are used when determining whether the item is present in the original list. Like **CONS**, **ADJOIN** has no effect on the original list--if you want to modify a particular list, you need to assign the value returned by **ADJOIN** to the place where the list came from. The modify macro **PUSHNEW** does this for you automatically.

```
CL-USER> (defparameter *set* ())
*SET*
CL-USER> (adjoin 1 *set*)
(1)
CL-USER> *set*
NIL
CL-USER> (setf *set* (adjoin 1 *set*))
(1)
CL-USER> (pushnew 2 *set*)
(2 1)
CL-USER> *set*
(2 1)
CL-USER> (pushnew 2 *set*)
(2 1)
```

You can test whether a given item is in a set with **MEMBER** and the related functions **MEMBER-IF** and **MEMBER-IF-NOT**. These functions are similar to the sequence functions **FIND**, **FIND-IF**, and **FIND-IF-NOT** except they can be used only with lists. And instead of returning the item when it's present, they return the cons cell containing the item--in other words, the sublist starting with the desired item. When the desired item isn't present in the list, all three functions return **NIL**.

The remaining set-theoretic functions provide bulk operations: **INTERSECTION**, **UNION**, **SET-DIFFERENCE**, and **SET-EXCLUSIVE-OR**. Each of these functions takes two lists and `:key` and `:test` keyword arguments and returns a new list representing the set resulting from performing the appropriate set-theoretic operation on the two lists: **INTERSECTION** returns a list containing all the elements found in both arguments. **UNION** returns a list containing one instance of each unique element from the two arguments.[3] **SET-DIFFERENCE** returns a list containing all the elements from the first argument that don't appear in the second argument. And **SET-EXCLUSIVE-OR** returns a list containing those elements appearing in only one or the other of the two argument lists but not in both. Each of these functions also has a

recycling counterpart whose name is the same except with an *N* prefix.

Finally, the function **SUBSETP** takes two lists and the usual `:key` and `:test` keyword arguments and returns true if the first list is a subset of the second--if every element in the first list is also present in the second list. The order of the elements in the lists doesn't matter.

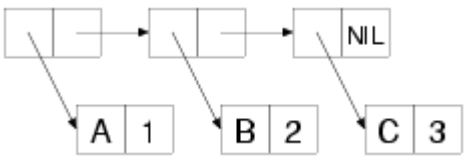
```
CL-USER> (subsetp '(3 2 1) '(1 2 3 4))
T
CL-USER> (subsetp '(1 2 3 4) '(3 2 1))
NIL
```

Lookup Tables: Alists and Plists

In addition to trees and sets, you can build tables that map keys to values out of cons cells. Two flavors of cons-based lookup tables are commonly used, both of which I've mentioned in passing in previous chapters. They're *association lists*, also called *alists*, and *property lists*, also known as *plists*. While you wouldn't use either alists or plists for large tables--for that you'd use a hash table--it's worth knowing how to work with them both because for small tables they can be more efficient than hash tables and because they have some useful properties of their own.

An alist is a data structure that maps keys to values and also supports reverse lookups, finding the key when given a value. Alists also support adding key/value mappings that shadow existing mappings in such a way that the shadowing mapping can later be removed and the original mappings exposed again.

Under the covers, an alist is essentially a list whose elements are themselves cons cells. Each element can be thought of as a key/value pair with the key in the cons cell's `CAR` and the value in the `CDR`. For instance, the following is a box-and-arrow diagram of an alist mapping the symbol `A` to the number 1, `B` to 2, and `C` to 3:



Unless the value in the **CDR** is a list, cons cells representing the key/value pairs will be *dotted pairs* in s-expression notation. The alist diagramed in the previous figure, for instance, is printed like this:

```
((A . 1) (B . 2) (C . 3))
```

The main lookup function for alists is **ASSOC**, which takes a key and an alist and returns the first cons cell whose **CAR** matches the key or **NIL** if no match is found.

```
CL-USER> (assoc 'a '((a . 1) (b . 2) (c . 3)))
(A . 1)
CL-USER> (assoc 'c '((a . 1) (b . 2) (c . 3)))
(C . 3)
CL-USER> (assoc 'd '((a . 1) (b . 2) (c . 3)))
NIL
```

To get the value corresponding to a given key, you simply pass the result of **ASSOC** to **CDR**.

```
CL-USER> (cdr (assoc 'a '((a . 1) (b . 2) (c . 3))))
1
```

By default the key given is compared to the keys in the alist using **EQL**, but you can change that with the standard combination of `:key` and `:test` keyword arguments. For instance, if you wanted to use string keys, you might write this:

```
CL-USER> (assoc "a" '(("a" . 1) ("b" . 2) ("c" . 3)) :test #'string=)
("a" . 1)
```

Without specifying `:test` to be **STRING=**, that **ASSOC** would probably return **NIL** because two strings with the same contents aren't necessarily **EQL**.

```
CL-USER> (assoc "a" '(("a" . 1) ("b" . 2) ("c" . 3)))
NIL
```

Because ASSOC searches the list by scanning from the front of the list, one key/value pair in an alist can shadow other pairs with the same key later in the list.

```
CL-USER> (assoc 'a '((a . 10) (a . 1) (b . 2) (c . 3)))
(A . 10)
```

You can add a pair to the front of an alist with CONS like this:

```
(cons (cons 'new-key 'new-value) alist)
```

However, as a convenience, Common Lisp provides the function ACONS, which lets you write this:

```
(acons 'new-key 'new-value alist)
```

Like CONS, ACONS is a function and thus can't modify the place holding the alist it's passed. If you want to modify an alist, you need to write either this:

```
(setf alist (acons 'new-key 'new-value alist))
```

or this:

```
(push (cons 'new-key 'new-value) alist)
```

Obviously, the time it takes to search an alist with ASSOC is a function of how deep in the list the matching pair is found. In the worst case, determining that no pair matches requires ASSOC to scan every element of the alist. However, since the basic mechanism for alists is so lightweight, for small tables an alist can outperform a hash table. Also, alists give you more flexibility in how you do the lookup. I already mentioned that ASSOC takes `:key` and `:test` keyword arguments. When those don't suit your needs, you may be able to use the ASSOC-IF and ASSOC-IF-NOT functions, which return the first

key/value pair whose CAR satisfies (or not, in the case of ASSOC-IF-NOT) the test function passed in the place of a specific item. And three functions--RASSOC, RASSOC-IF, and RASSOC-IF-NOT--work just like the corresponding ASSOC functions except they use the value in the CDR of each element as the key, performing a reverse lookup.

The function COPY-ALIST is similar to COPY-TREE except, instead of copying the whole tree structure, it copies only the cons cells that make up the list structure, plus the cons cells directly referenced from the CARs of those cells. In other words, the original alist and the copy will both contain the same objects as the keys and values, even if those keys or values happen to be made up of cons cells.

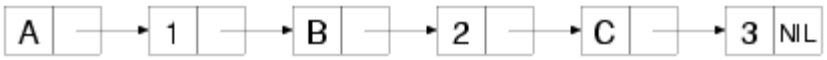
Finally, you can build an alist from two separate lists of keys and values with the function PAIRLIS. The resulting alist may contain the pairs either in the same order as the original lists or in reverse order. For example, you may get this result:

```
CL-USER> (pairlis '(a b c) '(1 2 3))
((C . 3) (B . 2) (A . 1))
```

Or you could just as well get this:

```
CL-USER> (pairlis '(a b c) '(1 2 3))
((A . 1) (B . 2) (C . 3))
```

The other kind of lookup table is the property list, or plist, which you used to represent the rows in the database in Chapter 3. Structurally a plist is just a regular list with the keys and values as alternating values. For instance, a plist mapping `A`, `B`, and `C`, to 1, 2, and 3 is simply the list `(A 1 B 2 C 3)`. In boxes-and-arrows form, it looks like this:



However, plists are less flexible than alists. In fact, plists support only one fundamental lookup operation, the function **GETF**, which takes a plist and a key and returns the associated value or **NIL** if the key isn't found. **GETF** also takes an optional third argument, which will be returned in place of **NIL** if the key isn't found.

Unlike **ASSOC**, which uses **EQL** as its default test and allows a different test function to be supplied with a `:test` argument, **GETF** always uses **EQ** to test whether the provided key matches the keys in the plist. Consequently, you should never use numbers or characters as keys in a plist; as you saw in Chapter 4, the behavior of **EQ** for those types is essentially undefined. Practically speaking, the keys in a plist are almost always symbols, which makes sense since plists were first invented to implement symbolic "properties," arbitrary mappings between names and values.

You can use **SETF** with **GETF** to set the value associated with a given key. **SETF** also treats **GETF** a bit specially in that the first argument to **GETF** is treated as the place to modify. Thus, you can use **SETF** of **GETF** to add a new key/value pair to an existing plist.

```
CL-USER> (defparameter *plist* ())
*PLIST*
CL-USER> *plist*
NIL
CL-USER> (setf (getf *plist* :a) 1)
1
CL-USER> *plist*
(:A 1)
CL-USER> (setf (getf *plist* :a) 2)
2
CL-USER> *plist*
(:A 2)
```

To remove a key/value pair from a plist, you use the macro **REMF**, which sets the place given as its first argument to a plist containing all the key/value pairs except the one specified. It returns true if the given key was actually found.

```
CL-USER> (remf *plist* :a)
T
CL-USER> *plist*
NIL
```

Like **GETF**, **REMF** always uses **EQ** to compare the given key to the keys in the plist.

Since plists are often used in situations where you want to extract several properties from the same plist, Common Lisp provides a function, **GET-PROPERTIES**, that makes it more efficient to extract multiple values from a single plist. It takes a plist and a list of keys to search for and returns, as multiple values, the first key found, the corresponding value, and the head of the list starting with the found key. This allows you to process a property list, extracting the desired properties, without continually rescanning from the front of the list. For instance, the following function efficiently processes--using the hypothetical function `process-property`--all the key/value pairs in a plist for a given list of keys:

```
(defun process-properties (plist keys)
  (loop while plist do
       (multiple-value-bind (key value tail) (get-properties plist keys)
         (when key (process-property key value))
         (setf plist (cddr tail)))))
```

The last special thing about plists is the relationship they have with symbols: every symbol object has an associated plist that can be used to store information about the symbol. The plist can be obtained via the function **SYMBOL-PLIST**. However, you rarely care

about the whole plist; more often you'll use the functions **GET**, which takes a symbol and a key and is shorthand for a **GETF** of the same key in the symbols **SYMBOL-PLIST**.

```
(get 'symbol 'key) === (getf (symbol-plist 'symbol) 'key)
```

Like **GETF**, **GET** is **SETF**able, so you can attach arbitrary information to a symbol like this:

```
(setf (get 'some-symbol 'my-key) "information")
```

To remove a property from a symbol's plist, you can use either **REMF** of **SYMBOL-PLIST** or the convenience function **REMPROP**.[4]

```
(remprop 'symbol 'key) === (remf (symbol-plist 'symbol key))
```

Being able to attach arbitrary information to names is quite handy when doing any kind of symbolic programming. For instance, one of the macros you'll write in Chapter 24 will attach information to names that other instances of the same macros will extract and use when generating their expansions.

DESTRUCTURING-BIND

One last tool for slicing and dicing lists that I need to cover since you'll need it in later chapters is the **DESTRUCTURING-BIND** macro. This macro provides a way to *destructure* arbitrary lists, similar to the way macro parameter lists can take apart their argument list. The basic skeleton of a **DESTRUCTURING-BIND** is as follows:

```
(destructuring-bind (parameter*) list
  body-form*)
```

The parameter list can include any of the types of parameters supported in macro parameter lists such as **&optional**, **&rest**, and **&key** parameters.[5] And, as in macro parameter lists, any parameter can be replaced with a nested destructuring parameter list, which takes apart the list that would otherwise have been bound to the replaced parameter. The *list* form is evaluated once and should return a list, which is then destructured and the appropriate values are bound to the variables in the parameter list. Then the *body-forms* are evaluated in order with those bindings in effect. Some simple examples follow:

```
(destructuring-bind (x y z) (list 1 2 3)
  (list :x x :y y :z z)) ==> (:X 1 :Y 2 :Z 3)

(destructuring-bind (x y z) (list 1 (list 2 20) 3)
  (list :x x :y y :z z)) ==> (:X 1 :Y (2 20) :Z 3)

(destructuring-bind (x (y1 y2) z) (list 1 (list 2 20) 3)
  (list :x x :y1 y1 :y2 y2 :z z)) ==> (:X 1 :Y1 2 :Y2 20 :Z 3)

(destructuring-bind (x (y1 &optional y2) z) (list 1 (list 2 20) 3)
  (list :x x :y1 y1 :y2 y2 :z z)) ==> (:X 1 :Y1 2 :Y2 20 :Z 3)

(destructuring-bind (x (y1 &optional y2) z) (list 1 (list 2) 3)
  (list :x x :y1 y1 :y2 y2 :z z)) ==> (:X 1 :Y1 2 :Y2 NIL :Z 3)

(destructuring-bind (&key x y z) (list :x 1 :y 2 :z 3)
  (list :x x :y y :z z)) ==> (:X 1 :Y 2 :Z 3)
```

```
(destructuring-bind (&key x y z) (list :z 1 :y 2 :x 3)
  (list :x x :y y :z z)) ==> (:X 3 :Y 2 :Z 1)
```

One kind of parameter you can use with **DESTRUCTURING-BIND** and also in macro parameter lists, though I didn't mention it in Chapter 8, is a **&whole** parameter. If specified, it must be the first parameter in a parameter list, and it's bound to the whole list form.[6] After a **&whole** parameter, other parameters can appear as usual and will extract specific parts of the list just as they would if the **&whole** parameter weren't there. An example of using **&whole** with **DESTRUCTURING-BIND** looks like this:

```
(destructuring-bind (&whole whole &key x y z) (list :z 1 :y 2 :x 3)
  (list :x x :y y :z z :whole whole))
==> (:X 3 :Y 2 :Z 1 :WHOLE (:Z 1 :Y 2 :X 3))
```

You'll use a **&whole** parameter in one of the macros that's part of the HTML generation library you'll develop in Chapter 31. However, I have a few more topics to cover before you can get to that. After two chapters on the rather Lispy topic of cons cells, you can now turn to the more prosaic matter of how to deal with files and filenames.

---

[1]It's possible to build a chain of cons cells where the **CDR** of the last cons cell isn't **NIL** but some other atom. This is called a *dotted* list because the last cons is a dotted pair.

[2]It may seem that the **NSUBST** family of functions can and in fact does modify the tree in place. However, there's one edge case: when the "tree" passed is, in fact, an atom, it can't be modified in place, so the result of **NSUBST** will be a different object than the argument: `(nsubst 'x 'y 'y) X`.

[3]**UNION** takes only one element from each list, but if either list contains duplicate elements, the result may also contain duplicates.

[4]It's also possible to directly **SETF SYMBOL-PLIST**. However, that's a bad idea, as different code may have added different properties to the symbol's plist for different reasons. If one

piece of code clobbers the symbol's whole plist, it may break other code that added its own properties to the plist.

[5]Macro parameter lists do support one parameter type, `&environment` parameters, which `DESTRUCTURING-BIND` doesn't. However, I didn't discuss that parameter type in Chapter 8, and you don't need to worry about it now either.

[6]When a `&whole` parameter is used in a macro parameter list, the form it's bound to is the whole macro form, including the name of the macro.

Copyright © 2003-2005, Peter Seibel

14. Files and File I/O

Common Lisp provides a rich library of functionality for dealing with files. In this chapter I'll focus on a few basic file-related tasks: reading and writing files and listing files in the file system. For these basic tasks, Common Lisp's I/O facilities are similar to those in other languages. Common Lisp provides a stream abstraction for reading and writing data and an abstraction, called *pathnames*, for manipulating filenames in an operating system-independent way. Additionally, Common Lisp provides other bits of functionality unique to Lisp such as the ability to read and write s-expressions.

Reading File Data

The most basic file I/O task is to read the contents of a file. You obtain a stream from which you can read a file's contents with the OPEN function. By default OPEN returns a character-based input stream you can pass to a variety of functions that read one or more characters of text: READ-CHAR reads a single character; READ-LINE reads a line of text, returning it as a string with the end-of-line character(s) removed; and READ reads a single s-expression, returning a Lisp object. When you're done with the stream, you can close it with the CLOSE function.

The only required argument to OPEN is the name of the file to read. As you'll see in the section "Filenames," Common Lisp provides a couple of ways to represent a filename, but the simplest is to use a string containing the name in the local file-naming syntax. So assuming that `/some/file/name.txt` is a file, you can open it like this:

```
(open "/some/file/name.txt")
```

You can use the object returned as the first argument to any of the read functions. For instance, to print the first line of the file, you can combine OPEN, READ-LINE, and CLOSE as follows:

```
(let ((in (open "/some/file/name.txt")))
  (format t "~a~%" (read-line in))
  (close in))
```

Of course, a number of things can go wrong while trying to open and read from a file. The file may not exist. Or you may unexpectedly hit the end of the file while reading. By default OPEN and the `READ-*` functions will signal an error in these situations. In Chapter 19, I'll

discuss how to recover from such errors. For now, however, there's a lighter-weight solution: each of these functions accepts arguments that modify its behavior in these exceptional situations.

If you want to open a possibly nonexistent file without **OPEN** signaling an error, you can use the keyword argument `:if-does-not-exist` to specify a different behavior. The three possible values are `:error`, the default; `:create`, which tells it to go ahead and create the file and then proceed as if it had already existed; and **NIL**, which tells it to return **NIL** instead of a stream. Thus, you can change the previous example to deal with the possibility that the file may not exist.

```
(let ((in (open "/some/file/name.txt" :if-does-not-exist nil)))
  (when in
    (format t "~a~%" (read-line in))
    (close in)))
```

The reading functions--**READ-CHAR**, **READ-LINE**, and **READ**--all take an optional argument, which defaults to true, that specifies whether they should signal an error if they're called at the end of the file. If that argument is **NIL**, they instead return the value of their third argument, which defaults to **NIL**. Thus, you could print all the lines in a file like this:

```
(let ((in (open "/some/file/name.txt" :if-does-not-exist nil)))
  (when in
    (loop for line = (read-line in nil)
         while line do (format t "~a~%" line))
    (close in)))
```

Of the three text-reading functions, **READ** is unique to Lisp. This is the same function that provides the *R* in the REPL and that's used to read Lisp source code. Each time it's called, it reads a single s-expression, skipping whitespace and comments, and returns the

Lisp object denoted by the s-expression. For instance, suppose `/some/file/name.txt` has the following contents:

```
(1 2 3)
456
"a string" ; this is a comment
((a b)
 (c d))
```

In other words, it contains four s-expressions: a list of numbers, a number, a string, and a list of lists. You can read those expressions like this:

```
CL-USER> (defparameter *s* (open "/some/file/name.txt"))
*S*
CL-USER> (read *s*)
(1 2 3)
CL-USER> (read *s*)
456
CL-USER> (read *s*)
"a string"
CL-USER> (read *s*)
((A B) (C D))
CL-USER> (close *s*)
T
```

As you saw in Chapter 3, you can use **`PRINT`** to print Lisp objects in "readable" form. Thus, whenever you need to store a bit of data in a file, **`PRINT`** and **`READ`** provide an easy way to do it without having to design a data format or write a parser. They even--as the previous example demonstrated--give you comments for free. And because s-expressions were designed to be human editable, it's also a fine format for things like configuration files.[1]

Reading Binary Data

By default **OPEN** returns character streams, which translate the underlying bytes to characters according to a particular character-encoding scheme.[2] To read the raw bytes, you need to pass **OPEN** an `:element-type` argument of `'(unsigned-byte 8)`.[3] You can pass the resulting stream to the function **READ-BYTE**, which will return an integer between 0 and 255 each time it's called. **READ-BYTE**, like the character-reading functions, also accepts optional arguments to specify whether it should signal an error if called at the end of the file and what value to return if not. In Chapter 24 you'll build a library that allows you to conveniently read structured binary data using **READ-BYTE**.[4]

Bulk Reads

One last reading function, **READ-SEQUENCE**, works with both character and binary streams. You pass it a sequence (typically a vector) and a stream, and it attempts to fill the sequence with data from the stream. It returns the index of the first element of the sequence that wasn't filled or the length of the sequence if it was able to completely fill it. You can also pass `:start` and `:end` keyword arguments to specify a subsequence that should be filled instead. The sequence argument must be a type that can hold elements of the stream's element type. Since most operating systems support some form of block I/O, **READ-SEQUENCE** is likely to be quite a bit more efficient than filling a sequence by repeatedly calling **READ-BYTE** or **READ-CHAR**.

File Output

To write data to a file, you need an output stream, which you obtain by calling **OPEN** with a `:direction` keyword argument of `:output`. When opening a file for output, **OPEN** assumes the file shouldn't already exist and will signal an error if it does. However, you can change that behavior with the `:if-exists` keyword argument. Passing the value `:supersede` tells **OPEN** to replace the existing file. Passing `:append` causes **OPEN** to open the existing file such that new data will be written at the end of the file, while `:overwrite` returns a stream that will overwrite existing data starting from the beginning of the file. And passing **NIL** will cause **OPEN** to return **NIL** instead of a stream if the file already exists. A typical use of **OPEN** for output looks like this:

```
(open "/some/file/name.txt" :direction :output :if-exists :supersede)
```

Common Lisp also provides several functions for writing data: **WRITE-CHAR** writes a single character to the stream. **WRITE-LINE** writes a string followed by a newline, which will be output as the appropriate end-of-line character or characters for the platform. Another function, **WRITE-STRING**, writes a string without adding any end-of-line characters. Two different functions can print just a newline: **TERPRI**--short for "terminate print"--unconditionally prints a newline character, and **FRESH-LINE** prints a newline character unless the stream is at the beginning of a line. **FRESH-LINE** is handy when you want to avoid spurious blank lines in textual output generated by different functions called in sequence. For example, suppose you have one function that generates output that should always be

followed by a line break and another that should start on a new line. But assume that if the functions are called one after the other, you don't want a blank line between the two bits of output. If you use **FRESH-LINE** at the beginning of the second function, its output will always start on a new line, but if it's called right after the first, it won't emit an extra line break.

Several functions output Lisp data as s-expressions: **PRINT** prints an s-expression preceded by an end-of-line and followed by a space. **PRIN1** prints just the s-expression. And the function **PPRINT** prints s-expressions like **PRINT** and **PRIN1** but using the "pretty printer," which tries to print its output in an aesthetically pleasing way.

However, not all objects can be printed in a form that **READ** will understand. The variable ***PRINT-READABLY*** controls what happens if you try to print such an object with **PRINT**, **PRIN1**, or **PPRINT**. When it's **NIL**, these functions will print the object in a special syntax that's guaranteed to cause **READ** to signal an error if it tries to read it; otherwise they will signal an error rather than print the object.

Another function, **PRINC**, also prints Lisp objects, but in a way designed for human consumption. For instance, **PRINC** prints strings without quotation marks. You can generate more elaborate text output with the incredibly flexible if somewhat arcane **FORMAT** function. I'll discuss some of the more important details of **FORMAT**, which essentially defines a mini-language for emitting formatted output, in Chapter 18.

To write binary data to a file, you have to **OPEN** the file with the same `:element-type` argument as you did to read it: `'(unsigned-byte 8)`.

You can then write individual bytes to the stream with `WRITE-BYTE`.

The bulk output function `WRITE-SEQUENCE` accepts both binary and character streams as long as all the elements of the sequence are of an appropriate type for the stream, either characters or bytes. As with `READ-SEQUENCE`, this function is likely to be quite a bit more efficient than writing the elements of the sequence one at a time.

Closing Files

As anyone who has written code that deals with lots of files knows, it's important to close files when you're done with them, because file handles tend to be a scarce resource. If you open files and don't close them, you'll soon discover you can't open any more files.[5] It might seem straightforward enough to just be sure every `OPEN` has a matching `CLOSE`. For instance, you could always structure your file using code like this:

```
(let ((stream (open "/some/file/name.txt")))
  ;; do stuff with stream
  (close stream))
```

However, this approach suffers from two problems. One is simply that it's error prone--if you forget the `CLOSE`, the code will leak a file handle every time it runs. The other--and more significant--problem is that there's no guarantee you'll get to the `CLOSE`. For instance, if the code prior to the `CLOSE` contains a `RETURN` or `RETURN-FROM`, you could leave the `LET` without closing the stream. Or, as you'll see in Chapter 19, if any of the code before the `CLOSE` signals an error, control may jump out of the `LET` to an error handler and never come back to close the stream.

Common Lisp provides a general solution to the problem of how to ensure that certain code always runs: the special operator `UNWIND-PROTECT`, which I'll discuss in Chapter 20. However, because the pattern of opening a file, doing something with the resulting stream, and then closing the stream is so common, Common Lisp

provides a macro, **WITH-OPEN-FILE**, built on top of **UNWIND-PROTECT**, to encapsulate this pattern. This is the basic form:

```
(with-open-file (stream-var open-argument*)
  body-form*)
```

The forms in *body-forms* are evaluated with *stream-var* bound to a file stream opened by a call to **OPEN** with *open-arguments* as its arguments. **WITH-OPEN-FILE** then ensures the stream in *stream-var* is closed before the **WITH-OPEN-FILE** form returns. Thus, you can write this to read a line from a file:

```
(with-open-file (stream "/some/file/name.txt")
  (format t "~a~%" (read-line stream)))
```

To create a new file, you can write something like this:

```
(with-open-file (stream "/some/file/name.txt" :direction :output)
  (format stream "Some text."))
```

You'll probably use **WITH-OPEN-FILE** for 90-99 percent of the file I/O you do--the only time you need to use raw **OPEN** and **CLOSE** calls is if you need to open a file in a function and keep the stream around after the function returns. In that case, you must take care to eventually close the stream yourself, or you'll leak file descriptors and may eventually end up unable to open any more files.

Filenames

So far you've used strings to represent filenames. However, using strings as filenames ties your code to a particular operating system and file system. Likewise, if you programmatically construct names according to the rules of a particular naming scheme (separating directories with /, say), you also tie your code to a particular file system.

To avoid this kind of nonportability, Common Lisp provides another representation of filenames: pathname objects. Pathnames represent filenames in a structured way that makes them easy to manipulate without tying them to a particular filename syntax. And the burden of translating back and forth between strings in the local syntax--called *namestrings*--and pathnames is placed on the Lisp implementation.

Unfortunately, as with many abstractions designed to hide the details of fundamentally different underlying systems, the pathname abstraction introduces its own complications. When pathnames were designed, the set of file systems in general use was quite a bit more variegated than those in common use today. Consequently, some nooks and crannies of the pathname abstraction make little sense if all you're concerned about is representing Unix or Windows filenames. However, once you understand which parts of the pathname abstraction you can ignore as artifacts of pathnames' evolutionary history, they do provide a convenient way to manipulate filenames.[6]

Most places a filename is called for, you can use either a namestring or a pathname. Which to use depends mostly on where the name originated. Filenames provided by the user--for example, as arguments or as values in configuration files--will typically be namestrings, since the user knows what operating system they're running on and shouldn't be expected to care about the details of how Lisp represents filenames. But programmatically generated filenames will be pathnames because you can create them portably. A stream returned by `OPEN` also represents a filename, namely, the filename that was originally used to open the stream. Together these three types are collectively referred to as *pathname designators*. All the built-in functions that expect a filename argument accept all three types of pathname designator. For instance, all the places in the previous section where you used a string to represent a filename, you could also have passed a pathname object or a stream.

How We Got Here

The historical diversity of file systems in existence during the 70s and 80s can be easy to forget. Kent Pitman, one of the principal technical editors of the Common Lisp standard, described the situation once in comp.lang.lisp (Message-ID: `sfwzo74np6w.fsf@world.std.com`) thusly:

> The dominant file systems at the time the design [of Common Lisp] was done were TOPS-10, TENEX, TOPS-20, VAX VMS, AT&T Unix, MIT Multics, MIT ITS, not to mention a bunch of mainframe [OSs]. Some were uppercase only, some mixed, some were case-sensitive but case- translating (like CL). Some had dirs as files, some not. Some had quote chars for funny file chars, some not. Some had wildcards, some didn't. Some had :up in relative pathnames, some didn't. Some had namable root dirs, some didn't. There were file systems with no directories, file systems with non-hierarchical directories, file systems with no file types, file systems with no versions, file systems with no devices, and so on.

If you look at the pathname abstraction from the point of view of any single file system, it seems baroque. However, if you take even two such similar file systems as Windows and Unix, you can already begin to see differences the pathname system can help abstract away--Windows filenames contain a drive letter, for instance, while Unix filenames don't. The other advantage of having the pathname abstraction designed to handle the wide variety of file systems that existed in the past is that it's more likely to be able to handle file systems that may exist in the future. If, say, versioning file systems come back into vogue, Common Lisp will be ready.

How Pathnames Represent Filenames

A pathname is a structured object that represents a filename using six components: host, device, directory, name, type, and version. Most of these components take on atomic values, usually strings; only the directory component is further structured, containing a list of directory names (as strings) prefaced with the keyword `:absolute` or `:relative`. However, not all pathname components are needed on all platforms--this is one of the reasons pathnames strike many new Lispers as gratuitously complex. On the other hand, you don't really need to worry about which components may or may not be used to represent names on a particular file system unless you need to create a new pathname object from scratch, which you'll almost never need to do. Instead, you'll usually get hold of pathname objects either by letting the implementation parse a file system-specific namestring into a pathname object or by creating a new pathname that takes most of its components from an existing pathname.

For instance, to translate a namestring to a pathname, you use the **`PATHNAME`** function. It takes a pathname designator and returns an equivalent pathname object. When the designator is already a pathname, it's simply returned. When it's a stream, the original filename is extracted and returned. When the designator is a namestring, however, it's parsed according to the local filename syntax. The language standard, as a platform-neutral document, doesn't specify any particular mapping from namestring to

pathname, but most implementations follow the same conventions on a given operating system.

On Unix file systems, only the directory, name, and type components are typically used. On Windows, one more component--usually the device or host--holds the drive letter. On these platforms, a namestring is parsed by first splitting it into elements on the path separator--a slash on Unix and a slash or backslash on Windows. The drive letter on Windows will be placed into either the device or the host component. All but the last of the other name elements are placed in a list starting with `:absolute` or `:relative` depending on whether the name (ignoring the drive letter, if any) began with a path separator. This list becomes the directory component of the pathname. The last element is then split on the rightmost dot, if any, and the two parts put into the name and type components of the pathname.[7]

You can examine these individual components of a pathname with the functions **`PATHNAME-DIRECTORY`**, **`PATHNAME-NAME`**, and **`PATHNAME-TYPE`**.

```
(pathname-directory (pathname "/foo/bar/baz.txt")) ==> (:ABSOLUTE "foo" "bar")
(pathname-name (pathname "/foo/bar/baz.txt"))      ==> "baz"
(pathname-type (pathname "/foo/bar/baz.txt"))      ==> "txt"
```

Three other functions--**`PATHNAME-HOST`**, **`PATHNAME-DEVICE`**, and **`PATHNAME-VERSION`**--allow you to get at the other three pathname components, though they're unlikely to have interesting values on Unix. On Windows either **`PATHNAME-HOST`** or **`PATHNAME-DEVICE`** will return the drive letter.

Like many other built-in objects, pathnames have their own read syntax, `#p` followed by a double-quoted string. This allows you to

print and read back s-expressions containing pathname objects, but because the syntax depends on the namestring parsing algorithm, such data isn't necessarily portable between operating systems.

```
(pathname "/foo/bar/baz.txt") ==> #p"/foo/bar/baz.txt"
```

To translate a pathname back to a namestring--for instance, to present to the user--you can use the function **NAMESTRING**, which takes a pathname designator and returns a namestring. Two other functions, **DIRECTORY-NAMESTRING** and **FILE-NAMESTRING**, return a partial namestring. **DIRECTORY-NAMESTRING** combines the elements of the directory component into a local directory name, and **FILE-NAMESTRING** combines the name and type components.[8]

```
(namestring #p"/foo/bar/baz.txt")           ==> "/foo/bar/baz.txt"
(directory-namestring #p"/foo/bar/baz.txt") ==> "/foo/bar/"
(file-namestring #p"/foo/bar/baz.txt")      ==> "baz.txt"
```

Constructing New Pathnames

You can construct arbitrary pathnames using the **MAKE-PATHNAME** function. It takes one keyword argument for each pathname component and returns a pathname with any supplied components filled in and the rest **NIL**.[9]

```
(make-pathname
  :directory '(:absolute "foo" "bar")
  :name "baz"
  :type "txt") ==> #p"/foo/bar/baz.txt"
```

However, if you want your programs to be portable, you probably don't want to make pathnames completely from scratch: even though the pathname abstraction protects you from unportable filename syntax, filenames can be unportable in other ways. For instance, the filename `/home/peter/foo.txt` is no good on an OS X box where `/home/` is called `/Users/`.

Another reason not to make pathnames completely from scratch is that different implementations use the pathname components slightly differently. For instance, as mentioned previously, some Windows-based Lisp implementations store the drive letter in the device component while others store it in the host component. If you write code like this:

```
(make-pathname :device "c" :directory '(:absolute "foo" "bar") :name "baz")
```

it will be correct on some implementations but not on others.

Rather than making names from scratch, you can build a new pathname based on an existing pathname with **MAKE-PATHNAME**'s keyword parameter `:defaults`. With this parameter you can provide

a pathname designator, which will supply the values for any components not specified by other arguments. For example, the following expression creates a pathname with an `.html` extension and all other components the same as the pathname in the variable `input-file`:

```
(make-pathname :type "html" :defaults input-file)
```

Assuming the value in `input-file` was a user-provided name, this code will be robust in the face of operating system and implementation differences such as whether filenames have drive letters in them and where they're stored in a pathname if they do.[10]

You can use the same technique to create a pathname with a different directory component.

```
(make-pathname :directory '(:relative "backups") :defaults input-file)
```

However, this will create a pathname whose whole directory component is the relative directory `backups/`, regardless of any directory component `input-file` may have had. For example:

```
(make-pathname :directory '(:relative "backups")
               :defaults #p"/foo/bar/baz.txt") ==> #p"backups/baz.txt"
```

Sometimes, though, you want to combine two pathnames, at least one of which has a relative directory component, by combining their directory components. For instance, suppose you have a relative pathname such as `#p"foo/bar.html"` that you want to combine with an absolute pathname such as `#p"/www/html/"` to get `#p"/www/html/foo/bar.html"`. In that case, **`MAKE-PATHNAME`** won't do; instead, you want **`MERGE-PATHNAMES`**.

MERGE-PATHNAMES takes two pathnames and merges them, filling in any **NIL** components in the first pathname with the corresponding value from the second pathname, much like **MAKE-PATHNAME** fills in any unspecified components with components from the `:defaults` argument. However, **MERGE-PATHNAMES** treats the directory component specially: if the first pathname's directory is relative, the directory component of the resulting pathname will be the first pathname's directory relative to the second pathname's directory. Thus:

```
(merge-pathnames #p"foo/bar.html" #p"/www/html/") ==> #p"/www/html/foo/bar.html"
```

The second pathname can also be relative, in which case the resulting pathname will also be relative.

```
(merge-pathnames #p"foo/bar.html" #p"html/") ==> #p"html/foo/bar.html"
```

To reverse this process and obtain a filename relative to a particular root directory, you can use the handy function **ENOUGH-NAMESTRING**.

```
(enough-namestring #p"/www/html/foo/bar.html" #p"/www/") ==> "html/foo/bar.html"
```

You can then combine **ENOUGH-NAMESTRING** with **MERGE-PATHNAMES** to create a pathname representing the same name but in a different root.

```
(merge-pathnames
  (enough-namestring #p"/www/html/foo/bar/baz.html" #p"/www/")
  #p"/www-backups/") ==> #p"/www-backups/html/foo/bar/baz.html"
```

MERGE-PATHNAMES is also used internally by the standard functions that actually access files in the file system to fill in incomplete pathnames. For instance, suppose you make a pathname with just a name and a type.

```
(make-pathname :name "foo" :type "txt") ==> #p"foo.txt"
```

If you try to use this pathname as an argument to **OPEN**, the missing components, such as the directory, must be filled in before Lisp will be able to translate the pathname to an actual filename. Common Lisp will obtain values for the missing components by merging the given pathname with the value of the variable ***DEFAULT-PATHNAME-DEFAULTS***. The initial value of this variable is determined by the implementation but is usually a pathname with a directory component representing the directory where Lisp was started and appropriate values for the host and device components, if needed. If invoked with just one argument, **MERGE-PATHNAMES** will merge the argument with the value of ***DEFAULT-PATHNAME-DEFAULTS***. For instance, if ***DEFAULT-PATHNAME-DEFAULTS*** is `#p"/home/peter/"`, then you'd get the following:

```
(merge-pathnames #p"foo.txt") ==> #p"/home/peter/foo.txt"
```

Two Representations of Directory Names

When dealing with pathnames that name directories, you need to be aware of one wrinkle. Pathnames separate the directory and name components, but Unix and Windows consider directories just another kind of file. Thus, on those systems, every directory has two different pathname representations.

One representation, which I'll call *file form*, treats a directory like any other file and puts the last element of the namestring into the name and type components. The other representation, *directory form*, places all the elements of the name in the directory component, leaving the name and type components **NIL**. If `/foo/bar/` is a directory, then both of the following pathnames name it.

```
(make-pathname :directory '(:absolute "foo") :name "bar") ; file form
(make-pathname :directory '(:absolute "foo" "bar"))       ; directory form
```

When you create pathnames with **MAKE-PATHNAME**, you can control which form you get, but you need to be careful when dealing with namestrings. All current implementations create file form pathnames unless the namestring ends with a path separator. But you can't rely on user-supplied namestrings necessarily being in one form or another. For instance, suppose you've prompted the user for a directory to save a file in and they entered `"/home/peter"`. If you pass that value as the `:defaults` argument of **MAKE-PATHNAME** like this:

```
(make-pathname :name "foo" :type "txt" :defaults user-supplied-name)
```

you'll end up saving the file in `/home/foo.txt` rather than the intended `/home/peter/foo.txt` because the `"peter"` in the namestring will be

placed in the name component when `user-supplied-name` is converted to a pathname. In the pathname portability library I'll discuss in the next chapter, you'll write a function called `pathname-as-directory` that converts a pathname to directory form. With that function you can reliably save the file in the directory indicated by the user.

```
(make-pathname
  :name "foo" :type "txt" :defaults (pathname-as-directory user-supplied-name))
```

Interacting with the File System

While the most common interaction with the file system is probably OPENing files for reading and writing, you'll also occasionally want to test whether a file exists, list the contents of a directory, delete and rename files, create directories, and get information about a file such as who owns it, when it was last modified, and its length. This is where the generality of the pathname abstraction begins to cause a bit of pain: because the language standard doesn't specify how functions that interact with the file system map to any specific file system, implementers are left with a fair bit of leeway.

That said, most of the functions that interact with the file system are still pretty straightforward. I'll discuss the standard functions here and point out the ones that suffer from nonportability between implementations. In the next chapter you'll develop a pathname portability library to smooth over some of those nonportability issues.

To test whether a file exists in the file system corresponding to a pathname designator--a pathname, namestring, or file stream--you can use the function PROBE-FILE. If the file named by the pathname designator exists, PROBE-FILE returns the file's *truename*, a pathname with any file system-level translations such as resolving symbolic links performed. Otherwise, it returns NIL. However, not all implementations support using this function to test whether a directory exists. Also, Common Lisp doesn't provide a portable way to test whether a given file that exists is a regular file or a directory. In the next chapter you'll wrap PROBE-FILE with a new function,

`file-exists-p`, that can both test whether a directory exists and tell you whether a given name is the name of a file or directory.

Similarly, the standard function for listing files in the file system, **`DIRECTORY`**, works fine for simple cases, but the differences between implementations make it tricky to use portably. In the next chapter you'll define a `list-directory` function that smoothes over some of these differences.

`DELETE-FILE` and **`RENAME-FILE`** do what their names suggest. **`DELETE-FILE`** takes a pathname designator and deletes the named file, returning true if it succeeds. Otherwise it signals a **`FILE-ERROR`**.[11]

`RENAME-FILE` takes two pathname designators and renames the file named by the first name to the second name.

You can create directories with the function **`ENSURE-DIRECTORIES-EXIST`**. It takes a pathname designator and ensures that all the elements of the directory component exist and are directories, creating them as necessary. It returns the pathname it was passed, which makes it convenient to use inline.

```
(with-open-file (out (ensure-directories-exist name) :direction :output)
  ...
  )
```

Note that if you pass **`ENSURE-DIRECTORIES-EXIST`** a directory name, it should be in directory form, or the leaf directory won't be created.

The functions **`FILE-WRITE-DATE`** and **`FILE-AUTHOR`** both take a pathname designator. **`FILE-WRITE-DATE`** returns the time in number of seconds since midnight January 1, 1900, Greenwich mean time

(GMT), that the file was last written, and **FILE-AUTHOR** returns, on Unix and Windows, the file owner.[12]

To find the length of a file, you can use the function **FILE-LENGTH**. For historical reasons **FILE-LENGTH** takes a stream as an argument rather than a pathname. In theory this allows **FILE-LENGTH** to return the length in terms of the element type of the stream. However, since on most present-day operating systems, the only information available about the length of a file, short of actually reading the whole file to measure it, is its length in bytes, that's what most implementations return, even when **FILE-LENGTH** is passed a character stream. However, the standard doesn't require this behavior, so for predictable results, the best way to get the length of a file is to use a binary stream.[13]

```
(with-open-file (in filename :element-type '(unsigned-byte 8))
  (file-length in))
```

A related function that also takes an open file stream as its argument is **FILE-POSITION**. When called with just a stream, this function returns the current position in the file--the number of elements that have been read from or written to the stream. When called with two arguments, the stream and a position designator, it sets the position of the stream to the designated position. The position designator must be the keyword `:start`, the keyword `:end`, or a non-negative integer. The two keywords set the position of the stream to the start or end of the file while an integer moves to the indicated position in the file. With a binary stream the position is simply a byte offset into the file. However, for character streams things are a bit more complicated because of character-encoding issues. Your best bet, if

you need to jump around within a file of textual data, is to only ever pass, as a second argument to the two-argument version of **FILE-POSITION**, a value previously returned by the one-argument version of **FILE-POSITION** with the same stream argument.

Other Kinds of I/O

In addition to file streams, Common Lisp supports other kinds of streams, which can also be used with the various reading, writing, and printing I/O functions. For instance, you can read data from, or write data to, a string using **STRING-STREAM**s, which you can create with the functions **MAKE-STRING-INPUT-STREAM** and **MAKE-STRING-OUTPUT-STREAM**.

MAKE-STRING-INPUT-STREAM takes a string and optional start and end indices to bound the area of the string from which data should be read and returns a character stream that you can pass to any of the character-based input functions such as **READ-CHAR**, **READ-LINE**, or **READ**. For example, if you have a string containing a floating-point literal in Common Lisp's syntax, you can convert it to a float like this:

```
(let ((s (make-string-input-stream "1.23")))
  (unwind-protect (read s)
    (close s)))
```

Similarly, **MAKE-STRING-OUTPUT-STREAM** creates a stream you can use with **FORMAT**, **PRINT**, **WRITE-CHAR**, **WRITE-LINE**, and so on. It takes no arguments. Whatever you write, a string output stream will be accumulated into a string that can then be obtained with the function **GET-OUTPUT-STREAM-STRING**. Each time you call **GET-OUTPUT-STREAM-STRING**, the stream's internal string is cleared so you can reuse an existing string output stream.

However, you'll rarely use these functions directly, because the macros **WITH-INPUT-FROM-STRING** and **WITH-OUTPUT-TO-STRING** provide a more convenient interface. **WITH-INPUT-FROM-STRING** is similar to

WITH-OPEN-FILE--it creates a string input stream from a given string and then executes the forms in its body with the stream bound to the variable you provide. For instance, instead of the **LET** form with the explicit **UNWIND-PROTECT**, you'd probably write this:

```
(with-input-from-string (s "1.23")
  (read s))
```

The **WITH-OUTPUT-TO-STRING** macro is similar: it binds a newly created string output stream to a variable you name and then executes its body. After all the body forms have been executed, **WITH-OUTPUT-TO-STRING** returns the value that would be returned by **GET-OUTPUT-STREAM-STRING**.

```
CL-USER> (with-output-to-string (out)
           (format out "hello, world ")
           (format out "~s" (list 1 2 3)))
"hello, world (1 2 3)"
```

The other kinds of streams defined in the language standard provide various kinds of stream "plumbing," allowing you to plug together streams in almost any configuration. A **BROADCAST-STREAM** is an output stream that sends any data written to it to a set of output streams provided as arguments to its constructor function, **MAKE-BROADCAST-STREAM**.[14] Conversely, a **CONCATENATED-STREAM** is an input stream that takes its input from a set of input streams, moving from stream to stream as it hits the end of each stream. **CONCATENATED-STREAM**s are constructed with the function **MAKE-CONCATENATED-STREAM**, which takes any number of input streams as arguments.

Two kinds of bidirectional streams that can plug together streams in a couple ways are **TWO-WAY-STREAM** and **ECHO-STREAM**. Their

constructor functions, **MAKE-TWO-WAY-STREAM** and **MAKE-ECHO-STREAM**, both take two arguments, an input stream and an output stream, and return a stream of the appropriate type, which you can use with both input and output functions.

In a **TWO-WAY-STREAM** every read you perform will return data read from the underlying input stream, and every write will send data to the underlying output stream. An **ECHO-STREAM** works essentially the same way except that all the data read from the underlying input stream is also echoed to the output stream. Thus, the output stream of an **ECHO-STREAM** stream will contain a transcript of both sides of the conversation.

Using these five kinds of streams, you can build almost any topology of stream plumbing you want.

Finally, although the Common Lisp standard doesn't say anything about networking APIs, most implementations support socket programming and typically implement sockets as another kind of stream, so you can use all the regular I/O functions with them.[15]

Now you're ready to move on to building a library that smoothes over some of the differences between how the basic pathname functions behave in different Common Lisp implementations.

---

[1]Note, however, that while the Lisp reader knows how to skip comments, it completely skips them. Thus, if you use **READ** to read in a configuration file containing comments and then use **PRINT** to save changes to the data, you'll lose the comments.

[2]By default **OPEN** uses the default character encoding for the operating system, but it also accepts a keyword parameter, `:external-format`, that can pass implementation-defined

values that specify a different encoding. Character streams also translate the platform-specific end-of-line sequence to the single character `#\Newline`.

[3]The type `(unsigned-byte 8)` indicates an 8-bit byte; Common Lisp "byte" types aren't a fixed size since Lisp has run at various times on architectures with byte sizes from 6 to 9 bits, to say nothing of the PDP-10, which had individually addressable variable-length bit fields of 1 to 36 bits.

[4]In general, a stream is either a character stream or a binary stream, so you can't mix calls to **`READ-BYTE`** and **`READ-CHAR`** or other character-based read functions. However, some implementations, such as Allegro, support so-called bivalent streams, which support both character and binary I/O.

[5]Some folks expect this wouldn't be a problem in a garbage-collected language such as Lisp. It is the case in most Lisp implementations that a stream that becomes garbage will automatically be closed. However, this isn't something to rely on--the problem is that garbage collectors usually run only when memory is low; they don't know about other scarce resources such as file handles. If there's plenty of memory available, it's easy to run out of file handles long before the garbage collector runs.

[6]Another reason the pathname system is considered somewhat baroque is because of the inclusion of *logical pathnames*. However, you can use the rest of the pathname system perfectly well without knowing anything more about logical pathnames than that you can safely ignore them. Briefly, logical pathnames allow Common Lisp programs to contain references to pathnames without naming specific files. Logical pathnames could then be mapped to specific locations in an actual file system when the program was installed by defining a "logical pathname translation" that translates logical pathnames matching certain wildcards to pathnames representing files in the file system, so-called physical pathnames. They have their uses in certain situations, but you can get pretty far without worrying about them.

[7]Many Unix-based implementations treat filenames whose last element starts with a dot and don't contain any other dots specially, putting the whole element, with the dot, in the name component and leaving the type component **`NIL`**.

```
(pathname-name (pathname "/foo/.emacs")) ==> ".emacs"
(pathname-type (pathname "/foo/.emacs")) ==> NIL
```

However, not all implementations follow this convention; some will create a pathname with "" as the name and `emacs` as the type.

[8]The name returned by **FILE-NAMESTRING** also includes the version component on file systems that use it.

[9]The host component may not default to **NIL**, but if not, it will be an opaque implementation-defined value.

[10]For absolutely maximum portability, you should really write this:

```
(make-pathname :type "html" :version :newest :defaults input-file)
```

Without the `:version` argument, on a file system with built-in versioning, the output pathname would inherit its version number from the input file which isn't likely to be right--if the input file has been saved many times it will have a much higher version number than the generated HTML file. On implementations without file versioning, the `:version` argument should be ignored. It's up to you if you care that much about portability.

[11]See Chapter 19 for more on handling errors.

[12]For applications that need access to other file attributes on a particular operating system or file system, libraries provide bindings to underlying C system calls. The Osicat library at `http://common-lisp.net/project/osicat/` provides a simple API built using the Universal Foreign Function Interface (UFFI), which should run on most Common Lisps that run on a POSIX operating system.

[13]The number of bytes and characters in a file can differ even if you're not using a multibyte character encoding. Because character streams also translate platform-specific line endings to a single `#\Newline` character, on Windows (which uses CRLF as its line ending) the number of characters will typically be smaller than the number of bytes. If you really have to know the number of characters in a file, you have to bite the bullet and write something like this:

```
(with-open-file (in filename)
  (loop while (read-char in nil) count t))
```

or maybe something more efficient like this:

```
(with-open-file (in filename)
  (let ((scratch (make-string 4096)))
    (loop for read = (read-sequence scratch in)
          while (plusp read) sum read)))
```

[14]**MAKE-BROADCAST-STREAM** can make a data black hole by calling it with no arguments.

[15]The biggest missing piece in Common Lisp's standard I/O facilities is a way for users to define new stream classes. There are, however, two de facto standards for user-defined streams. During the Common Lisp standardization, David Gray of Texas Instruments wrote a draft proposal for an API to allow users to define new stream classes. Unfortunately, there wasn't time to work out all the issues raised by his draft to include it in the language standard. However, many implementations support some form of so-called Gray Streams, basing their API on Gray's draft proposal. Another, newer API, called Simple Streams, has been developed by Franz and included in Allegro Common Lisp. It was designed to improve the performance of user-defined streams relative to Gray Streams and has been adopted by some of the open-source Common Lisp implementations.

Copyright © 2003-2005, Peter Seibel

15. Practical: A Portable Pathname Library

As I discussed in the previous chapter, Common Lisp provides an abstraction, the pathname, that's supposed to insulate you from the details of how different operating systems and file systems name files. Pathnames provide a useful API for manipulating names as names, but when it comes to the functions that actually interact with the file system, things get a bit hairy.

The root of the problem, as I mentioned, is that the pathname abstraction was designed to represent filenames on a much wider variety of file systems than are commonly used now. Unfortunately, by making pathnames abstract enough to account for a wide variety of file systems, Common Lisp's designers left implementers with a fair number of choices to make about how exactly to map the pathname abstraction onto any particular file system. Consequently, different implementers, each implementing the pathname abstraction for the same file system, just by making different choices at a few key junctions, could end up with conforming implementations that nonetheless provide different behavior for several of the main pathname-related functions.

However, one way or another, all implementations provide the same basic functionality, so it's not too hard to write a library that provides a consistent interface for the most common operations across different implementations. That's your task for this chapter. In

addition to giving you several useful functions that you'll use in future chapters, writing this library will give you a chance to learn how to write code that deals with differences between implementations.

The API

The basic operations the library will support will be getting a list of files in a directory and determining whether a file or directory with a given name exists. You'll also write a function for recursively walking a directory hierarchy, calling a given function for each pathname in the tree.

In theory, these directory listing and file existence operations are already provided by the standard functions `DIRECTORY` and `PROBE-FILE`. However, as you'll see, there are enough different ways to implement these functions--all within the bounds of valid interpretations of the language standard--that you'll want to write new functions that provide a consistent behavior across implementations.

FEATURES and Read-Time Conditionalization

Before you can implement this API in a library that will run correctly on multiple Common Lisp implementations, I need to show you the mechanism for writing implementation-specific code.

While most of the code you write can be "portable" in the sense that it will run the same on any conforming Common Lisp implementation, you may occasionally need to rely on implementation-specific functionality or to write slightly different bits of code for different implementations. To allow you to do so without totally destroying the portability of your code, Common Lisp provides a mechanism, called *read-time conditionalization*, that allows you to conditionally include code based on various features such as what implementation it's being run in.

The mechanism consists of a variable **`*FEATURES*`** and two extra bits of syntax understood by the Lisp reader. **`*FEATURES*`** is a list of symbols; each symbol represents a "feature" that's present in the implementation or on the underlying platform. These symbols are then used in *feature expressions* that evaluate to true or false depending on whether the symbols in the expression are present in **`*FEATURES*`**. The simplest feature expression is a single symbol; the expression is true if the symbol is in **`*FEATURES*`** and false if it isn't. Other feature expressions are boolean expressions built out of **`NOT`**, **`AND`**, and **`OR`** operators. For instance, if you wanted to conditionalize some code to be included only if the features `foo` and `bar` were present, you could write the feature expression `(and foo bar)`.

The reader uses feature expressions in conjunction with two bits of syntax, `#+` and `#-`. When the reader sees either of these bits of syntax, it first reads a feature expression and then evaluates it as I just described. When a feature expression following a `#+` is true, the reader reads the next expression normally. Otherwise it skips the next expression, treating it as whitespace. `#-` works the same way except it reads the form if the feature expression is false and skips it if it's true.

The initial value of **`*FEATURES*`** is implementation dependent, and what functionality is implied by the presence of any given symbol is likewise defined by the implementation. However, all implementations include at least one symbol that indicates what implementation it is. For instance, Allegro Common Lisp includes the symbol `:allegro`, CLISP includes `:clisp`, SBCL includes `:sbcl`, and CMUCL includes `:cmu`. To avoid dependencies on packages that may or may not exist in different implementations, the symbols in **`*FEATURES*`** are usually keywords, and the reader binds **`*PACKAGE*`** to the **`KEYWORD`** package while reading feature expressions. Thus, a name with no package qualification will be read as a keyword symbol. So, you could write a function that behaves slightly differently in each of the implementations just mentioned like this:

```
(defun foo ()
  #+allegro (do-one-thing)
  #+sbcl (do-another-thing)
  #+clisp (something-else)
  #+cmu (yet-another-version)
  #-(or allegro sbcl clisp cmu) (error "Not implemented"))
```

In Allegro that code will be read as if it had been written like this:

```
(defun foo ()
  (do-one-thing))
```

while in SBCL the reader will read this:

```
(defun foo ()
  (do-another-thing))
```

while in an implementation other than one of the ones specifically conditionalized, it will read this:

```
(defun foo ()
  (error "Not implemented"))
```

Because the conditionalization happens in the reader, the compiler doesn't even see expressions that are skipped.[1] This means you pay no runtime cost for having different versions for different implementations. Also, when the reader skips conditionalized expressions, it doesn't bother interning symbols, so the skipped expressions can safely contain symbols from packages that may not exist in other implementations.

Packaging the Library

Speaking of packages, if you download the complete code for this library, you'll see that it's defined in a new package, `com.gigamonkeys.pathnames`. I'll discuss the details of defining and using packages in Chapter 21. For now you should note that some implementations provide their own packages that contain functions with some of the same names as the ones you'll define in this chapter and make those names available in the CL-USER package. Thus, if you try to define the functions from this library while in the CL-USER package, you may get errors or warnings about clobbering existing definitions. To avoid this possibility, you can create a file called `packages.lisp` with the following contents:

```
(in-package :cl-user)

(defpackage :com.gigamonkeys.pathnames
  (:use :common-lisp)
  (:export
```

```
      :list-directory
      :file-exists-p
      :directory-pathname-p
      :file-pathname-p
      :pathname-as-directory
      :pathname-as-file
      :walk-directory
      :directory-p
      :file-p))
```

and LOAD it. Then at the REPL or at the top of the file where you type the definitions from this chapter, type the following expression:

```
(in-package :com.gigamonkeys.pathnames)
```

In addition to avoiding name conflicts with symbols already available in CL-USER, packaging the library this way also makes it easier to use in other code, as you'll see in several future chapters.

Listing a Directory

You can implement the function for listing a single directory, `list-directory`, as a thin wrapper around the standard function **DIRECTORY**. **DIRECTORY** takes a special kind of pathname, called a *wild pathname*, that has one or more components containing the special value `:wild` and returns a list of pathnames representing files in the file system that match the wild pathname.[2] The matching algorithm--like most things having to do with the interaction between Lisp and a particular file system--isn't defined by the language standard, but most implementations on Unix and Windows follow the same basic scheme.

The **DIRECTORY** function has two problems that you need to address with `list-directory`. The main one is that certain aspects of its behavior differ fairly significantly between different Common Lisp implementations, even on the same operating system. The other is that while **DIRECTORY** provides a powerful interface for listing files, to use it properly requires understanding some rather subtle points about the pathname abstraction. Between these subtleties and the idiosyncrasies of different implementations, actually writing portable code that uses **DIRECTORY** to do something as simple as listing all the files and subdirectories in a single directory can be a frustrating experience. You can deal with those subtleties and idiosyncrasies once and for all, by writing `list-directory`, and forget them thereafter.

One subtlety I discussed in Chapter 14 is the two ways to represent the name of a directory as a pathname: directory form and file form.

To get **DIRECTORY** to return a list of files in `/home/peter/`, you need to pass it a wild pathname whose directory component is the directory you want to list and whose name and type components are `:wild`. Thus, to get a listing of the files in `/home/peter/`, it might seem you could write this:

```
(directory (make-pathname :name :wild :type :wild :defaults home-dir))
```

where `home-dir` is a pathname representing `/home/peter/`. This would work if `home-dir` were in directory form. But if it were in file form--for example, if it had been created by parsing the namestring `"/home/peter"`--then that same expression would list all the files in `/home` since the name component `"peter"` would be replaced with `:wild`.

To avoid having to worry about explicitly converting between representations, you can define `list-directory` to accept a nonwild pathname in either form, which it will then convert to the appropriate wild pathname.

To help with this, you should define a few helper functions. One, `component-present-p`, will test whether a given component of a pathname is "present," meaning neither **NIL** nor the special value `:unspecific`.[3] Another, `directory-pathname-p`, tests whether a pathname is already in directory form, and the third, `pathname-as-directory`, converts any pathname to a directory form pathname.

```
(defun component-present-p (value)
  (and value (not (eql value :unspecific))))

(defun directory-pathname-p  (p)
  (and
   (not (component-present-p (pathname-name p)))
```

```
   (not (component-present-p (pathname-type p)))
   p))

(defun pathname-as-directory (name)
  (let ((pathname (pathname name)))
    (when (wild-pathname-p pathname)
      (error "Can't reliably convert wild pathnames."))
    (if (not (directory-pathname-p name))
      (make-pathname
       :directory (append (or (pathname-directory pathname) (list :relative))
                          (list (file-namestring pathname)))
       :name      nil
       :type      nil
       :defaults pathname)
      pathname)))
```

Now it seems you could generate a wild pathname to pass to **DIRECTORY** by calling **MAKE-PATHNAME** with a directory form name returned by `pathname-as-directory`. Unfortunately, it's not quite that simple, thanks to a quirk in CLISP's implementation of **DIRECTORY**. In CLISP, **DIRECTORY** won't return files with no extension unless the type component of the wildcard is **NIL** rather than `:wild`. So you can define a function, `directory-wildcard`, that takes a pathname in either directory or file form and returns a proper wildcard for the given implementation using read-time conditionalization to make a pathname with a `:wild` type component in all implementations except for CLISP and **NIL** in CLISP.

```
(defun directory-wildcard (dirname)
  (make-pathname
   :name :wild
   :type #-clisp :wild #+clisp nil
   :defaults (pathname-as-directory dirname)))
```

Note how each read-time conditional operates at the level of a single expression After `#-clisp`, the expression `:wild` is either read or skipped; likewise, after `#+clisp`, the **NIL** is read or skipped.

Now you can take a first crack at the `list-directory` function.

```
(defun list-directory (dirname)
  (when (wild-pathname-p dirname)
```

```
    (error "Can only list concrete directory names."))
  (directory (directory-wildcard dirname)))
```

As it stands, this function would work in SBCL, CMUCL, and LispWorks. Unfortunately, a couple more implementation differences remain to be smoothed over. One is that not all implementations will return subdirectories of the given directory. Allegro, SBCL, CMUCL, and LispWorks do. OpenMCL doesn't by default but will if you pass **DIRECTORY** a true value via the implementation-specific keyword argument `:directories`. CLISP's **DIRECTORY** returns subdirectories only when it's passed a wildcard pathname with `:wild` as the last element of the directory component and **NIL** name and type components. In this case, it returns *only* subdirectories, so you'll need to call **DIRECTORY** twice with different wildcards and combine the results.

Once you get all the implementations returning directories, you'll discover they can also differ in whether they return the names of directories in directory or file form. You want `list-directory` to always return directory names in directory form so you can differentiate subdirectories from regular files based on just the name. Except for Allegro, all the implementations this library will support do that. Allegro, on the other hand, requires you to pass **DIRECTORY** the implementation-specific keyword argument `:directories-are-files` **NIL** to get it to return directories in file form.

Once you know how to make each implementation do what you want, actually writing `list-directory` is simply a matter of combining the different versions using read-time conditionals.

```
(defun list-directory (dirname)
  (when (wild-pathname-p dirname)
```

```
    (error "Can only list concrete directory names."))
  (let ((wildcard (directory-wildcard dirname)))

    #+(or sbcl cmu lispworks)
    (directory wildcard)

    #+openmcl
    (directory wildcard :directories t)

    #+allegro
    (directory wildcard :directories-are-files nil)

    #+clisp
    (nconc
     (directory wildcard)
     (directory (clisp-subdirectories-wildcard wildcard)))

    #-(or sbcl cmu lispworks openmcl allegro clisp)
    (error "list-directory not implemented")))
```

The function `clisp-subdirectories-wildcard` isn't actually specific to CLISP, but since it isn't needed by any other implementation, you can guard its definition with a read-time conditional. In this case, since the expression following the `#+` is the whole **`DEFUN`**, the whole function definition will be included or not, depending on whether `clisp` is present in **`*FEATURES*`**.

```
#+clisp
(defun clisp-subdirectories-wildcard (wildcard)
  (make-pathname
   :directory (append (pathname-directory wildcard) (list :wild))
   :name nil
   :type nil
   :defaults wildcard))
```

Testing a File's Existence

To replace **PROBE-FILE**, you can define a function called `file-exists-p`. It should accept a pathname and return an equivalent pathname if the file exists and **NIL** if it doesn't. It should be able to accept the name of a directory in either directory or file form but should always return a directory form pathname if the file exists and is a directory. This will allow you to use `file-exists-p`, along with `directory-pathname-p`, to test whether an arbitrary name is the name of a file or directory.

In theory, `file-exists-p` is quite similar to the standard function **PROBE-FILE**; indeed, in several implementations--SBCL, LispWorks, and OpenMCL--**PROBE-FILE** already gives you the behavior you want for `file-exists-p`. But not all implementations of **PROBE-FILE** behave quite the same.

Allegro and CMUCL's **PROBE-FILE** functions are close to what you need--they will accept the name of a directory in either form but, instead of returning a directory form name, simply return the name in the same form as the argument it was passed. Luckily, if passed the name of a nondirectory in directory form, they return **NIL**. So with those implementations you can get the behavior you want by first passing the name to **PROBE-FILE** in directory form--if the file exists and is a directory, it will return the directory form name. If that call returns **NIL**, then you try again with a file form name.

CLISP, on the other hand, once again has its own way of doing things. Its **PROBE-FILE** immediately signals an error if passed a name

in directory form, regardless of whether a file or directory exists with that name. It also signals an error if passed a name in file form that's actually the name of a directory. For testing whether a directory exists, CLISP provides its own function: `probe-directory` (in the `ext` package). This is almost the mirror image of **`PROBE-FILE`**: it signals an error if passed a name in file form or if passed a name in directory form that happens to name a file. The only difference is it returns **`T`** rather than a pathname when the named directory exists.

But even in CLISP you can implement the desired semantics by wrapping the calls to **`PROBE-FILE`** and `probe-directory` in **`IGNORE-ERRORS`**.[4]

```
(defun file-exists-p (pathname)
  #+(or sbcl lispworks openmcl)
  (probe-file pathname)

  #+(or allegro cmu)
  (or (probe-file (pathname-as-directory pathname))
      (probe-file pathname))

  #+clisp
  (or (ignore-errors
        (probe-file (pathname-as-file pathname)))
      (ignore-errors
        (let ((directory-form (pathname-as-directory pathname)))
          (when (ext:probe-directory directory-form)
            directory-form))))

  #-(or sbcl cmu lispworks openmcl allegro clisp)
  (error "file-exists-p not implemented"))
```

The function `pathname-as-file` that you need for the CLISP implementation of `file-exists-p` is the inverse of the previously defined `pathname-as-directory`, returning a pathname that's the file form equivalent of its argument. This function, despite being needed here only by CLISP, is generally useful, so define it for all implementations and make it part of the library.

```
(defun pathname-as-file (name)
  (let ((pathname (pathname name)))
    (when (wild-pathname-p pathname)
      (error "Can't reliably convert wild pathnames."))
    (if (directory-pathname-p name)
      (let* ((directory (pathname-directory pathname))
             (name-and-type (pathname (first (last directory)))))
        (make-pathname
         :directory (butlast directory)
         :name (pathname-name name-and-type)
         :type (pathname-type name-and-type)
         :defaults pathname))
      pathname)))
```

Walking a Directory Tree

Finally, to round out this library, you can implement a function called `walk-directory`. Unlike the functions defined previously, this function doesn't need to do much of anything to smooth over implementation differences; it just needs to use the functions you've already defined. However, it's quite handy, and you'll use it several times in subsequent chapters. It will take the name of a directory and a function and call the function on the pathnames of all the files under the directory, recursively. It will also take two keyword arguments: `:directories` and `:test`. When `:directories` is true, it will call the function on the pathnames of directories as well as regular files. The `:test` argument, if provided, specifies another function that's invoked on each pathname before the main function is; the main function will be called only if the test function returns true.

```
(defun walk-directory (dirname fn &key directories (test (constantly t)))
  (labels
      ((walk (name)
         (cond
           ((directory-pathname-p name)
            (when (and directories (funcall test name))
              (funcall fn name))
            (dolist (x (list-directory name)) (walk x)))
           ((funcall test name) (funcall fn name)))))
    (walk (pathname-as-directory dirname))))
```

Now you have a useful library of functions for dealing with pathnames. As I mentioned, these functions will come in handy in later chapters, particularly Chapters 23 and 27, where you'll use `walk-directory` to crawl through directory trees containing spam messages and MP3 files. But before we get to that, though, I need to talk about object orientation, the topic of the next two chapters.

[1]One slightly annoying consequence of the way read-time conditionalization works is that there's no easy way to write a fall-through case. For example, if you add support for another implementation to `foo` by adding another expression guarded with `#+`, you need to remember to also add the same feature to the `or` feature expression after the `#-` or the **ERROR** form will be evaluated after your new code runs.

[2]Another special value, `:wild-inferiors`, can appear as part of the directory component of a wild pathname, but you won't need it in this chapter.

[3]Implementations are allowed to return `:unspecific` instead of **NIL** as the value of pathname components in certain situations such as when the component isn't used by that implementation.

[4]This is slightly broken in the sense that if **PROBE-FILE** signals an error for some other reason, this code will interpret it incorrectly. Unfortunately, the CLISP documentation doesn't specify what errors might be signaled by **PROBE-FILE** and `probe-directory`, and experimentation seems to show that they signal `simple-file-error`s in most erroneous situations.

Copyright © 2003-2005, Peter Seibel

16. Object Reorientation: Generic Functions

Because the invention of Lisp predated the rise of object-oriented programming by a couple decades,[1] new Lispers are sometimes surprised to discover what a thoroughly object-oriented language Common Lisp is. Common Lisp's immediate predecessors were developed at a time when object orientation was an exciting new idea and there were many experiments with ways to incorporate the ideas of object orientation, especially as manifested in Smalltalk, into Lisp. As part of the Common Lisp standardization, a synthesis of several of these experiments emerged under the name Common Lisp Object System, or CLOS. The ANSI standard incorporated CLOS into the language, so it no longer really makes sense to speak of CLOS as a separate entity.

The features CLOS contributed to Common Lisp range from those that can hardly be avoided to relatively esoteric manifestations of Lisp's language-as-language-building-tool philosophy. Complete coverage of all these features is beyond the scope of this book, but in this chapter and the next I'll describe the bread-and-butter features and give an overview of Common Lisp's approach to objects.

You should note at the outset that Common Lisp's object system offers a fairly different embodiment of the principles of object orientation than many other languages. If you have a deep

understanding of the fundamental ideas behind object orientation, you'll likely appreciate the particularly powerful and general way Common Lisp manifests those ideas. On the other hand, if your experience with object orientation has been largely with a single language, you may find Common Lisp's approach somewhat foreign; you should try to avoid assuming that there's only one way for a language to support object orientation.[2] If you have little object-oriented programming experience, you should have no trouble understanding the explanations here, though it may help to ignore the occasional comparisons to the way other languages do things.

Generic Functions and Classes

The fundamental idea of object orientation is that a powerful way to organize a program is to define data types and then associate operations with those data types. In particular, you want to be able to invoke an operation and have the exact behavior determined by the type of the object or objects on which the operation was invoked. The classic example used, seemingly by all introductions to object orientation, is an operation `draw` that can be applied to objects representing various geometric shapes. Different implementations of the `draw` operation can be provided for drawing circles, triangles, and squares, and a call to `draw` will actually result in drawing a circle, triangle, or square, depending on the type of the object to which the `draw` operation is applied. The different implementations of `draw` are defined separately, and new versions can be defined that draw other shapes without having to change the code of either the caller or any of the other `draw` implementations. This feature of object orientation goes by the fancy Greek name *polymorphism*, meaning "many forms," because a single conceptual operation, such as drawing an object, can take many different concrete forms.

Common Lisp, like most object-oriented languages today, is class-based; all objects are *instances* of a particular class.[3] The class of an object determines its representation--built-in classes such as **`NUMBER`** and **`STRING`** have opaque representations accessible only via the standard functions for manipulating those types, while instances of user-defined classes, as you'll see in the next chapter, consist of named parts called *slots*.

Classes are arranged in a hierarchy, a taxonomy for all objects. A class can be defined as a *subclass* of other classes, called its *superclasses*. A class *inherits* part of its definition from its superclasses and instances of a class are also considered instances of the superclasses. In Common Lisp, the hierarchy of classes has a single root, the class `T`, which is a direct or indirect superclass of every other class. Thus, every datum in Common Lisp is an instance of `T`.[4] Common Lisp also supports *multiple inheritance*--a single class can have multiple direct superclasses.

Outside the Lisp family, almost all object-oriented languages follow the basic pattern established by Simula of having behavior associated with classes through *methods* or *member functions* that belong to a particular class. In these languages, a method is invoked on a particular object, and the class of that object determines what code runs. This model of method invocation is called--after the Smalltalk terminology--*message passing*. Conceptually, method invocation in a message-passing system starts by sending a *message* containing the name of the method to run and any arguments to the object on which the method is being invoked. The object then uses its class to look up the method associated with the name in the message and runs it. Because each class can have its own method for a given name, the same message, sent to different objects, can invoke different methods.

Early Lisp object systems worked in a similar way, providing a special function `SEND` that could be used to send a message to a particular object. However, this wasn't entirely satisfactory, as it

made method invocations different from normal function calls. Syntactically method invocations were written like this:

```
(send object 'foo)
```

rather than like this:

```
(foo object)
```

More significantly, because methods weren't functions, they couldn't be passed as arguments to higher-order functions such as **MAPCAR**; if one wanted to call a method on all the elements of a list with **MAPCAR**, one had to write this:

```
(mapcar #'(lambda (object) (send object 'foo)) objects)
```

rather than this:

```
(mapcar #'foo objects)
```

Eventually the folks working on Lisp object systems unified methods with functions by creating a new kind of function called a *generic function*. In addition to solving the problems just described, generic functions opened up new possibilities for the object system, including many features that simply don't make sense in a message-passing object system.

Generic functions are the heart of Common Lisp's object system and the topic of the rest of this chapter. While I can't talk about generic functions without some mention of classes, for now I'll focus on how to define and use generic functions. In the next chapter I'll show you how to define your own classes.

Generic Functions and Methods

A generic function defines an abstract operation, specifying its name and a parameter list but no implementation. Here, for example, is how you might define a generic function, `draw`, that will be used to draw different kinds of shapes on the screen:

```
(defgeneric draw (shape)
  (:documentation "Draw the given shape on the screen."))
```

I'll discuss the syntax of **`DEFGENERIC`** in the next section; for now just note that this definition doesn't contain any actual code.

A generic function is generic in the sense that it can--at least in theory--accept any objects as arguments.[5] However, by itself a generic function can't actually do anything; if you just define a generic function, no matter what arguments you call it with, it will signal an error. The actual implementation of a generic function is provided by *methods*. Each method provides an implementation of the generic function for particular classes of arguments. Perhaps the biggest difference between a generic function-based system and a message-passing system is that methods don't belong to classes; they belong to the generic function, which is responsible for determining what method or methods to run in response to a particular invocation.

Methods indicate what kinds of arguments they can handle by *specializing* the required parameters defined by the generic function. For instance, on the generic function `draw`, you might define one method that specializes the `shape` parameter for objects that are

instances of the class `circle` while another method specializes `shape` for objects that are instances of the class `triangle`. They would look like this, eliding the actual drawing code:

```
(defmethod draw ((shape circle))
  ...)

(defmethod draw ((shape triangle))
  ...)
```

When a generic function is invoked, it compares the actual arguments it was passed with the specializers of each of its methods to find the *applicable* methods--those methods whose specializers are compatible with the actual arguments. If you invoke `draw`, passing an instance of `circle`, the method that specialized `shape` on the class `circle` is applicable, while if you pass it a `triangle`, then the method that specializes `shape` on the class `triangle` applies. In simple cases, only one method will be applicable, and it will handle the invocation. In more complex cases, there may be multiple methods that apply; they're then combined, as I'll discuss in the section "Method Combination," into a single *effective method* that handles the invocation.

You can specialize a parameter in two ways--usually you'll specify a class that the argument must be an instance of. Because instances of a class are also considered instances of that class's superclasses, a method with a parameter specialized on a particular class can be applicable whenever the corresponding argument is a direct instance of the specializing class or of any of its subclasses. The other kind of specializer is a so-called **`EQL`** specializer, which specifies a particular object to which the method applies.

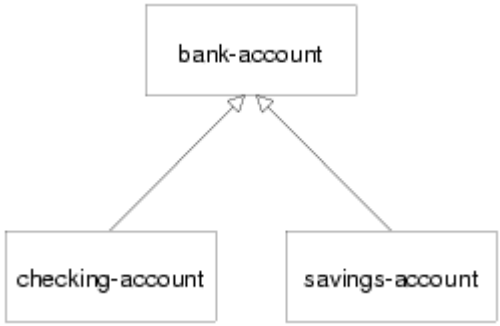
When a generic function has only methods specialized on a single parameter and all the specializers are class specializers, the result of invoking a generic function is quite similar to the result of invoking a method in a message-passing system--the combination of the name of the operation and the class of the object on which it's invoked determines what method to run.

However, reversing the order of lookup opens up possibilities not found in message-passing systems. Generic functions support methods that specialize on multiple parameters, provide a framework that makes multiple inheritance much more manageable, and let you use declarative constructs to control how methods are combined into an effective method, supporting several common usage patterns without a lot of boilerplate code. I'll discuss those topics in a moment. But first you need to look at the basics of the two macros used to define the generic functions `DEFGENERIC` and `DEFMETHOD`.

DEFGENERIC

To give you a feel for these macros and the various facilities they support, I'll show you some code you might write as part of a banking application--or, rather, a toy banking application; the point is to look at a few language features, not to learn how to really write banking software. For instance, this code doesn't even pretend to deal with such issues as multiple currencies let alone audit trails and transactional integrity.

Because I'm not going to discuss how to define new classes until the next chapter, for now you can just assume that certain classes already exist: for starters, assume there's a class `bank-account` and that it has two subclasses, `checking-account` and `savings-account`. The class hierarchy looks like this:



The first generic function will be `withdraw`, which decreases the account balance by a specified amount. If the balance is less than the amount, it should signal an error and leave the balance unchanged. You can start by defining the generic function with **`DEFGENERIC`**.

The basic form of **DEFGENERIC** is similar to **DEFUN** except with no body. The parameter list of **DEFGENERIC** specifies the parameters that must be accepted by all the methods that will be defined on the generic function. In the place of the body, a **DEFGENERIC** can contain various options. One option you should always include is `:documentation`, which you use to provide a string describing the purpose of the generic function. Because a generic function is purely abstract, it's important to be clear to both users and implementers what it's for. Thus, you might define `withdraw` like this:

```
(defgeneric withdraw (account amount)
  (:documentation "Withdraw the specified amount from the account.
Signal an error if the current balance is less than amount."))
```

DEFMETHOD

Now you're ready to use **DEFMETHOD** to define methods that implement `withdraw`.[6]

A method's parameter list must be *congruent* with its generic function's. In this case, that means all methods defined on `withdraw` must have exactly two required parameters. More generally, methods must have the same number of required and optional parameters and must be capable of accepting any arguments corresponding to any **&rest** or **&key** parameters specified by the generic function.[7]

Since the basics of withdrawing are the same for all accounts, you can define a method that specializes the `account` parameter on the `bank-account` class. You can assume the function `balance` returns the current balance of the account and can be used with **SETF**--and thus with **DECF**--to set the balance. The function **ERROR** is a standard function used to signal an error, which I'll discuss in greater detail in Chapter 19. Using those two functions, you can define a basic `withdraw` method that looks like this:

```
(defmethod withdraw ((account bank-account) amount)
  (when (< (balance account) amount)
    (error "Account overdrawn."))
  (decf (balance account) amount))
```

As this code suggests, the form of **DEFMETHOD** is even more like that of **DEFUN** than **DEFGENERIC**'s is. The only difference is that the required parameters can be specialized by replacing the parameter name with a two-element list. The first element is the name of the

parameter, and the second element is the specializer, either the name of a class or an **EQL** specializer, the form of which I'll discuss in a moment. The parameter name can be anything--it doesn't have to match the name used in the generic function, though it often will.

This method will apply whenever the first argument to `withdraw` is an instance of `bank-account`. The second parameter, `amount`, is implicitly specialized on **T**, and since all objects are instances of **T**, it doesn't affect the applicability of the method.

Now suppose all checking accounts have overdraft protection. That is, each checking account is linked to another bank account that's drawn upon when the balance of the checking account itself can't cover a withdrawal. You can assume that the function `overdraft-account` takes a `checking-account` object and returns a `bank-account` object representing the linked account.

Thus, withdrawing from a `checking-account` object requires a few extra steps compared to withdrawing from a standard `bank-account` object. You must first check whether the amount being withdrawn is greater than the account's current balance and, if it is, transfer the difference from the overdraft account. Then you can proceed as with a standard `bank-account` object.

So what you'd like to do is define a method on `withdraw` that specializes on `checking-account` to handle the transfer and then lets the method specialized on `bank-account` take control. Such a method might look like this:

```
(defmethod withdraw ((account checking-account) amount)
  (let ((overdraft (- amount (balance account))))
    (when (plusp overdraft)
```

```
      (withdraw (overdraft-account account) overdraft)
      (incf (balance account) overdraft)))
  (call-next-method))
```

The function **CALL-NEXT-METHOD** is part of the generic function machinery used to combine applicable methods. It indicates that control should be passed from this method to the method specialized on `bank-account`.[8] When it's called with no arguments, as it is here, the next method is invoked with whatever arguments were originally passed to the generic function. It can also be called with arguments, which will then be passed onto the next method.

You aren't required to invoke **CALL-NEXT-METHOD** in every method. However, if you don't, the new method is then responsible for completely implementing the desired behavior of the generic function. For example, if you had a subclass of `bank-account`, `proxy-account`, that didn't actually keep track of its own balance but instead delegated withdrawals to another account, you might write a method like this (assuming a function, `proxied-account`, that returns the proxied account):

```
(defmethod withdraw ((proxy proxy-account) amount)
  (withdraw (proxied-account proxy) amount))
```

Finally, **DEFMETHOD** also allows you to create methods specialized on a particular object with an **EQL** specializer. For example, suppose the banking app is going to be deployed in a particularly corrupt bank. Suppose the variable `*account-of-bank-president*` holds a reference to a particular bank account that belongs--as the name suggests--to the bank's president. Further suppose the variable `*bank*` represents the bank as a whole, and the function `embezzle`

steals money from the bank. The bank president might ask you to "fix" `withdraw` to handle his account specially.

```
(defmethod withdraw ((account (eql *account-of-bank-president*)) amount)
  (let ((overdraft (- amount (balance account))))
    (when (plusp overdraft)
      (incf (balance account) (embezzle *bank* overdraft)))
  (call-next-method)))
```

Note, however, that the form in the **EQL** specializer that provides the object to specialize on--`*account-of-bank-president*` in this case--is evaluated once, when the **DEFMETHOD** is evaluated. This method will be specialized on the value of `*account-of-bank-president*` at the time the method is defined; changing the variable later won't change the method.

Method Combination

Outside the body of a method, `CALL-NEXT-METHOD` has no meaning. Within a method, it's given a meaning by the generic function machinery that builds an *effective method* each time the generic function is invoked using all the methods applicable to that particular invocation. This notion of building an effective method by combining applicable methods is the heart of the generic function concept and is the thing that allows generic functions to support facilities not found in message-passing systems. So it's worth taking a closer look at what's really happening. Folks with the message-passing model deeply ingrained in their consciousness should pay particular attention because generic functions turn method dispatching inside out compared to message passing, making the generic function, rather than the class, the prime mover.

Conceptually, the effective method is built in three steps: First, the generic function builds a list of applicable methods based on the actual arguments it was passed. Second, the list of applicable methods is sorted according to the *specificity* of their parameter specializers. Finally, methods are taken in order from the sorted list and their code combined to produce the effective method.[9]

To find applicable methods, the generic function compares the actual arguments with the corresponding parameter specializers in each of its methods. A method is applicable if, and only if, all the specializers are compatible with the corresponding arguments.

When the specializer is the name of a class, it's compatible if it names the actual class of the argument or one of its superclasses. (Recall that parameters without explicit specializers are implicitly specialized on the class `T` so will be compatible with any argument.) An `EQL` specializer is compatible only when the argument is the same object as was specified in the specializer.

Because *all* the arguments are checked against the corresponding specializers, they all affect whether a method is applicable. Methods that explicitly specialize more than one parameter are called *multimethods*; I'll discuss them in the section "Multimethods."

After the applicable methods have been found, the generic function machinery needs to sort them before it can combine them into an effective method. To order two applicable methods, the generic function compares their parameter specializers from left to right,[10] and the first specializer that's different between the two methods determines their ordering, with the method with the more specific specializer coming first.

Because only applicable methods are being sorted, you know all class specializers will name classes that the corresponding argument is actually an instance of. In the typical case, if two class specializers differ, one will be a subclass of the other. In that case, the specializer naming the subclass is considered more specific. This is why the method that specialized `account` on `checking-account` was considered more specific than the method that specialized it on `bank-account`.

Multiple inheritance slightly complicates the notion of specificity since the actual argument may be an instance of two classes, neither of which is a subclass of the other. If such classes are used as parameter specializers, the generic function can't order them using only the rule that subclasses are more specific than their superclasses. In the next chapter I'll discuss how the notion of specificity is extended to deal with multiple inheritance. For now, suffice it to say that there's a deterministic algorithm for ordering class specializers.

Finally, an `EQL` specializer is always more specific than any class specializer, and because only applicable methods are being considered, if more than one method has an `EQL` specializer for a particular parameter, they must all have the same `EQL` specializer. The comparison of those methods will thus be decided based on other parameters.

The Standard Method Combination

Now that you understand how the applicable methods are found and sorted, you're ready to take a closer look at the last step--how the sorted list of methods is combined into a single effective method. By default, generic functions use what's called the *standard method combination*. The standard method combination combines methods so that **CALL-NEXT-METHOD** works as you've already seen--the most specific method runs first, and each method can pass control to the next most specific method via **CALL-NEXT-METHOD**.

However, there's a bit more to it than that. The methods I've been discussing so far are called *primary methods*. Primary methods, as their name suggests, are responsible for providing the primary implementation of a generic function. The standard method combination also supports three kinds of *auxiliary* methods: `:before`, `:after`, and `:around` methods. An auxiliary method definition is written with **DEFMETHOD** like a primary method but with a *method qualifier,* which names the type of method, between the name of the method and the parameter list. For instance, a `:before` method on `withdraw` that specializes the `account` parameter on the class `bank-account` would start like this:

```
(defmethod withdraw :before ((account bank-account) amount) ...)
```

Each kind of auxiliary method is combined into the effective method in a different way. All the applicable `:before` methods--not just the most specific--are run as part of the effective method. They run, as their name suggests, before the most specific primary method and

are run in most-specific-first order. Thus, `:before` methods can be used to do any preparation needed to ensure that the primary method can run. For instance, you could've used a `:before` method specialized on `checking-account` to implement the overdraft protection on checking accounts like this:

```
(defmethod withdraw :before ((account checking-account) amount)
  (let ((overdraft (- amount (balance account))))
    (when (plusp overdraft)
      (withdraw (overdraft-account account) overdraft)
      (incf (balance account) overdraft))))
```

This `:before` method has three advantages over a primary method. One is that it makes it immediately obvious how the method changes the overall behavior of the `withdraw` function--it's not going to interfere with the main behavior or change the result returned.

The next advantage is that a primary method specialized on a class more specific than `checking-account` won't interfere with this `:before` method, making it easier for an author of a subclass of `checking-account` to extend the behavior of `withdraw` while keeping part of the old behavior.

Lastly, since a `:before` method doesn't have to call **`CALL-NEXT-METHOD`** to pass control to the remaining methods, it's impossible to introduce a bug by forgetting to.

The other auxiliary methods also fit into the effective method in ways suggested by their names. All the `:after` methods run after the primary methods in most-specific-last order, that is, the reverse of the `:before` methods. Thus, the `:before` and `:after` methods combine to create a sort of nested wrapping around the core functionality provided by the primary methods--each more-specific

`:before` method will get a chance to set things up so the less-specific `:before` methods and primary methods can run successfully, and each more-specific `:after` method will get a chance to clean up after all the primary methods and less-specific `:after` methods.

Finally, `:around` methods are combined much like primary methods except they're run "around" all the other methods. That is, the code from the most specific `:around` method is run before anything else. Within the body of an `:around` method, **`CALL-NEXT-METHOD`** will lead to the code of the next most specific `:around` method or, in the least specific `:around` method, to the complex of `:before`, primary, and `:after` methods. Almost all `:around` methods will contain such a call to **`CALL-NEXT-METHOD`** because an `:around` method that doesn't will completely hijack the implementation of the generic function from all the methods except for more-specific `:around` methods.

Occasionally that kind of hijacking is called for, but typically `:around` methods are used to establish some dynamic context in which the rest of the methods will run--to bind a dynamic variable, for example, or to establish an error handler (as I'll discuss in Chapter 19). About the only time it's appropriate for an `:around` method to not call **`CALL-NEXT-METHOD`** is when it returns a result cached from a previous call to **`CALL-NEXT-METHOD`**. At any rate, an `:around` method that doesn't call **`CALL-NEXT-METHOD`** is responsible for correctly implementing the semantics of the generic function for all classes of arguments to which the method may apply, including future subclasses.

Auxiliary methods are just a convenient way to express certain common patterns more concisely and concretely. They don't actually

allow you to do anything you couldn't do by combining primary methods with diligent adherence to a few coding conventions and some extra typing. Perhaps their biggest benefit is that they provide a uniform framework for extending generic functions. Often a library will define a generic function and provide a default primary method, allowing users of the library to customize its behavior by defining appropriate auxiliary methods.

Other Method Combinations

In addition to the standard method combination, the language specifies nine other built-in method combinations known as the *simple* built-in method combinations. You can also define custom method combinations, though that's a fairly esoteric feature and beyond the scope of this book. I'll briefly cover how to use the simple built-in combinations to give you a sense of the possibilities.

All the simple combinations follow the same pattern: instead of invoking the most specific primary method and letting it invoke less-specific primary methods via `CALL-NEXT-METHOD`, the simple method combinations produce an effective method that contains the code of all the primary methods, one after another, all wrapped in a call to the function, macro, or special operator that gives the method combination its name. The nine combinations are named for the operators: `+`, `AND`, `OR`, `LIST`, `APPEND`, `NCONC`, `MIN`, `MAX`, and `PROGN`. The simple combinations also support only two kinds of methods, primary methods, which are combined as just described, and `:around` methods, which work like `:around` methods in the standard method combination.

For example, a generic function that uses the `+` method combination will return the sum of all the results returned by its primary methods. Note that the `AND` and `OR` method combinations won't necessarily run all the primary methods because of those macros' short-circuiting behavior--a generic function using the `AND` combination will return `NIL` as soon as one of the methods does and will return the value of the

last method otherwise. Similarly, the OR combination will return the first non-NIL value returned by any of the methods.

To define a generic function that uses a particular method combination, you include a `:method-combination` option in the DEFGENERIC form. The value supplied with this option is the name of the method combination you want to use. For example, to define a generic function, `priority`, that returns the sum of values returned by individual methods using the + method combination, you might write this:

```
(defgeneric priority (job)
  (:documentation "Return the priority at which the job should be run.")
  (:method-combination +))
```

By default all these method combinations combine the primary methods in most-specific-first order. However, you can reverse the order by including the keyword `:most-specific-last` after the name of the method combination in the DEFGENERIC form. The order probably doesn't matter if you're using the + combination unless the methods have side effects, but for demonstration purposes you can change `priority` to use most-specific-last order like this:

```
(defgeneric priority (job)
  (:documentation "Return the priority at which the job should be run.")
  (:method-combination + :most-specific-last))
```

The primary methods on a generic function that uses one of these combinations must be qualified with the name of the method combination. Thus, a primary method defined on `priority` might look like this:

```
(defmethod priority + ((job express-job)) 10)
```

This makes it obvious when you see a method definition that it's part of a particular kind of generic function.

All the simple built-in method combinations also support `:around` methods that work like `:around` methods in the standard method combination: the most specific `:around` method runs before any other methods, and **CALL-NEXT-METHOD** is used to pass control to less-and-less-specific `:around` methods until it reaches the combined primary methods. The `:most-specific-last` option doesn't affect the order of `:around` methods. And, as I mentioned before, the built-in method combinations don't support `:before` or `:after` methods.

Like the standard method combination, these method combinations don't allow you to do anything you couldn't do "by hand." Rather, they allow you to express *what* you want and let the language take care of wiring everything together for you, making your code both more concise and more expressive.

That said, probably 99 percent of the time, the standard method combination will be exactly what you want. Of the remaining 1 percent, probably 99 percent of them will be handled by one of the simple built-in method combinations. If you run into one of the 1 percent of 1 percent of cases where none of the built-in combinations suffices, you can look up **DEFINE-METHOD-COMBINATION** in your favorite Common Lisp reference.

Multimethods

Methods that explicitly specialize more than one of the generic function's required parameters are called *multimethods*. Multimethods are where generic functions and message passing really part ways. Multimethods don't fit into message-passing languages because they don't belong to a particular class; instead, each multimethod defines a part of the implementations of a given generic function that applies when the generic function is invoked with arguments that match *all* the method's specialized parameters.

Multimethods vs. Method Overloading

Programmers used to statically typed message-passing languages such as Java and C++ may think multimethods sound similar to a feature of those languages called *method overloading*. However, these two language features are actually quite different since overloaded methods are chosen at compile time, based on the compile-time type of the arguments, not at runtime. To see how this works, consider the following two Java classes:

```
public class A {
  public void foo(A a) { System.out.println("A/A"); }
  public void foo(B b) { System.out.println("A/B"); }
}

public class B extends A {
  public void foo(A a) { System.out.println("B/A"); }
  public void foo(B b) { System.out.println("B/B"); }
}
```

Now consider what happens when you run the `main` method from this class.

```
public class Main {
  public static void main(String[] argv) {
    A obj = argv[0].equals("A") ? new A() : new B();
    obj.foo(obj);
  }
}
```

When you tell `Main` to instantiate an `A`, it prints "A/A" as you'd probably expect.

```
bash$ java com.gigamonkeys.Main A
A/A
```

However, if you tell `Main` to instantiate a `B`, then the true type of `obj` is taken into account for only half the dispatching.

```
bash$ java com.gigamonkeys.Main B
B/A
```

If overloaded methods worked like Common Lisp's multimethods, then that would print "B/B" instead. It is possible to implement multiple dispatch by hand in message-passing languages, but this runs against the grain of the message-passing model since the code in a multiply dispatched method doesn't belong to any one class.

Multimethods are perfect for all those situations where, in a message-passing language, you struggle to decide to which class a certain behavior ought to belong. Is the sound a drum makes when it's hit with a drumstick a function of what kind of drum it is or what kind of stick you use to hit it? Both, of course. To model this situation in Common Lisp, you simply define a generic function `beat` that takes two arguments.

```
(defgeneric beat (drum stick)
  (:documentation
   "Produce a sound by hitting the given drum with the given stick."))
```

Then you can define various multimethods to implement `beat` for the combinations you care about. For example:

```
(defmethod beat ((drum snare-drum) (stick wooden-drumstick)) ...)
(defmethod beat ((drum snare-drum) (stick brush)) ...)
(defmethod beat ((drum snare-drum) (stick soft-mallet)) ...)
(defmethod beat ((drum tom-tom) (stick wooden-drumstick)) ...)
(defmethod beat ((drum tom-tom) (stick brush)) ...)
(defmethod beat ((drum tom-tom) (stick soft-mallet)) ...)
```

Multimethods don't help with the combinatorial explosion--if you need to model five kinds of drums and six kinds of sticks, and every combination makes a different sound, there's no way around it; you need thirty different methods to implement all the combinations, with or without multimethods. What multimethods do save you from is having to write a bunch of dispatching code by letting you use the same built-in polymorphic dispatching that's so useful when dealing with methods specialized on a single parameter.[11]

Multimethods also save you from having to tightly couple one set of classes with the other. In the drum/stick example, nothing requires the implementation of the drum classes to know about the various classes of drumstick, and nothing requires the drumstick classes to know anything about the various classes of drum. The multimethods connect the otherwise independent classes to describe their joint behavior without requiring any cooperation from the classes themselves.

To Be Continued . . .

I've covered the basics--and a bit beyond--of generic functions, the verbs of Common Lisp's object system. In the next chapter I'll show you how to define your own classes.

---

[1]The language now generally considered the first object-oriented language, Simula, was invented in the early 1960s, only a few years after McCarthy's first Lisp. However, object orientation didn't really take off until the 1980s when the first widely available version of Smalltalk was released, followed by the release of C++ a few years later. Smalltalk took quite a bit of inspiration from Lisp and combined it with ideas from Simula to produce a dynamic object-oriented language, while C++ combined Simula with C, another fairly static language, to yield a static object-oriented language. This early split has led to much confusion in the definition of object orientation. Folks who come from the C++ tradition tend to consider certain aspects of C++, such as strict data encapsulation, to be key characteristics of object orientation. Folks from the Smalltalk tradition, however, consider many features of C++ to be just that, features of C++, and not core to object orientation. Indeed, Alan Kay, the inventor of Smalltalk, is reported to have said, "I invented the term *object oriented*, and I can tell you that C++ wasn't what I had in mind."

[2]There are those who reject the notion that Common Lisp is in fact object oriented at all. In particular, folks who consider strict data encapsulation a key characteristic of object orientation--usually advocates of relatively static languages such as C++, Eiffel, or Java--don't consider Common Lisp to be properly object oriented. Of course, by that definition, Smalltalk, arguably one of the original and purest object-oriented languages, isn't object oriented either. On the other hand, folks who consider message passing to be the key to object orientation will also not be happy with the claim that Common Lisp is object oriented since Common Lisp's generic function orientation provides degrees of freedom not offered by pure message passing.

[3]Prototype-based languages are the other style of object-oriented language. In these languages, JavaScript being perhaps the most famous example, objects are created by cloning a prototypical object. The clone can then be modified and used as a prototype for other objects.

[4] **T** the constant value and **T** the class have no particular relationship except they happen to have the same name. **T** the value is a direct instance of the class **SYMBOL** and only indirectly an instance of **T** the class.

[5] Here, as elsewhere, *object* means any Lisp datum--Common Lisp doesn't distinguish, as some languages do, between objects and "primitive" data types; all data in Common Lisp are objects, and every object is an instance of a class.

[6] Technically you could skip the **DEFGENERIC** altogether--if you define a method with **DEFMETHOD** and no such generic function has been defined, one is automatically created. But it's good form to define generic functions explicitly, if only because it gives you a good place to document the intended behavior.

[7] A method can "accept" **&key** and **&rest** arguments defined in its generic function by having a **&rest** parameter, by having the same **&key** parameters, or by specifying **&allow-other-keys** along with **&key**. A method can also specify **&key** parameters not found in the generic function's parameter list--when the generic function is called, any **&key** parameter specified by the generic function or any applicable method will be accepted.

[8] **CALL-NEXT-METHOD** is roughly analogous to invoking a method on `super` in Java or using an explicitly class-qualified method or function name in Python or C++.

[9] While building the effective method sounds time-consuming, quite a bit of the effort in developing fast Common Lisp implementations has gone into making it efficient. One strategy is to cache the effective method so future calls with the same argument types will be able to proceed directly.

[10] Actually, the order in which specializers are compared is customizable via the **DEFGENERIC** option `:argument-precedence-order`, though that option is rarely used.

[11] In languages without multimethods, you must write dispatching code yourself to implement behavior that depends on the class of more than one object. The purpose of the popular Visitor design pattern is to structure a series of singly dispatched method calls so as to provide multiple dispatch. However, it requires one set of classes to know about the other. The Visitor pattern also quickly bogs down in a combinatorial explosion of dispatching methods if it's used to dispatch on more than two objects.

Copyright © 2003-2005, Peter Seibel

17. Object Reorientation: Classes

If generic functions are the verbs of the object system, classes are the nouns. As I mentioned in the previous chapter, all values in a Common Lisp program are instances of some class. Furthermore, all classes are organized into a single hierarchy rooted at the class `T`.

The class hierarchy consists of two major families of classes, built-in and user-defined classes. Classes that represent the data types you've been learning about up until now, classes such as `INTEGER`, `STRING`, and `LIST`, are all built-in. They live in their own section of the class hierarchy, arranged into appropriate sub- and superclass relationships, and are manipulated by the functions I've been discussing for much of the book up until now. You can't subclass these classes, but, as you saw in the previous chapter, you can define methods that specialize on them, effectively extending the behavior of those classes.[1]

But when you want to create new nouns--for instance, the classes used in the previous chapter for representing bank accounts--you need to define your own classes. That's the subject of this chapter.

DEFCLASS

You create user-defined classes with the **DEFCLASS** macro. Because behaviors are associated with a class by defining generic functions and methods specialized on the class, **DEFCLASS** is responsible only for defining the class as a data type.

The three facets of the class as a data type are its name, its relation to other classes, and the names of the slots that make up instances of the class.[2] The basic form of a **DEFCLASS** is quite simple.

```
(defclass name (direct-superclass-name*)
  (slot-specifier*))
```

What Are "User-Defined Classes"?

The term *user-defined classes* isn't a term from the language standard--technically what I'm talking about when I say *user-defined classes* are classes that subclass **STANDARD-OBJECT** and whose metaclass is **STANDARD-CLASS**. But since I'm not going to talk about the ways you can define classes that don't subclass **STANDARD-OBJECT** and whose metaclass isn't **STANDARD-CLASS**, you don't really have to worry about that. *User-defined* isn't a perfect term for these classes since the implementation may define certain classes the same way. However, to call them *standard* classes would be even more confusing since the built-in classes, such as **INTEGER** and **STRING**, are just as standard, if not more so, because they're defined by the language standard but they don't extend **STANDARD-OBJECT**. To further complicate matters, it's also possible for users to define new classes that *don't* subclass **STANDARD-OBJECT**. In particular, the macro **DEFSTRUCT** also defines new classes. But that's largely for backward compatibility--**DEFSTRUCT** predated CLOS and was retrofitted to define classes when CLOS was integrated into the language. But the classes it creates are fairly limited compared to **DEFCLASS**ed classes. So in this chapter I'll be discussing only classes defined with **DEFCLASS** that use the default metaclass of

STANDARD-CLASS, and I'll refer to them as *user-defined* for lack of a better term.

As with functions and variables, you can use any symbol as the name of a new class.[3] Class names are in a separate namespace from both functions and variables, so you can have a class, function, and variable all with the same name. You'll use the class name as the argument to **MAKE-INSTANCE**, the function that creates new instances of user-defined classes.

The *direct-superclass-names* specify the classes of which the new class is a subclass. If no superclasses are listed, the new class will directly subclass **STANDARD-OBJECT**. Any classes listed must be other user-defined classes, which ensures that each new class is ultimately descended from **STANDARD-OBJECT**. **STANDARD-OBJECT** in turn subclasses **T**, so all user-defined classes are part of the single class hierarchy that also contains all the built-in classes.

Eliding the slot specifiers for a moment, the **DEFCLASS** forms of some of the classes you used in the previous chapter might look like this:

```
(defclass bank-account () ...)

(defclass checking-account (bank-account) ...)

(defclass savings-account (bank-account) ...)
```

I'll discuss in the section "Multiple Inheritance" what it means to list more than one direct superclass in *direct-superclass-names*.

Slot Specifiers

The bulk of a **DEFCLASS** form consists of the list of slot specifiers. Each slot specifier defines a slot that will be part of each instance of the class. Each slot in an instance is a place that can hold a value, which can be accessed using the **SLOT-VALUE** function. **SLOT-VALUE** takes an object and the name of a slot as arguments and returns the value of the named slot in the given object. It can be used with **SETF** to set the value of a slot in an object.

A class also inherits slot specifiers from its superclasses, so the set of slots actually present in any object is the union of all the slots specified in a class's **DEFCLASS** form and those specified in all its superclasses.

At the minimum, a slot specifier names the slot, in which case the slot specifier can be just a name. For instance, you could define a `bank-account` class with two slots, `customer-name` and `balance`, like this:

```
(defclass bank-account ()
  (customer-name
   balance))
```

Each instance of this class will contain two slots, one to hold the name of the customer the account belongs to and another to hold the current balance. With this definition, you can create new `bank-account` objects using **MAKE-INSTANCE**.

```
(make-instance 'bank-account) ==> #<BANK-ACCOUNT @ #x724b93ba>
```

The argument to **MAKE-INSTANCE** is the name of the class to instantiate, and the value returned is the new object.[4] The printed representation of an object is determined by the generic function **PRINT-OBJECT**. In this case, the applicable method will be one provided by the implementation, specialized on **STANDARD-OBJECT**. Since not every object can be printed so that it can be read back, the **STANDARD-OBJECT** print method uses the `#<>` syntax, which will cause the reader to signal an error if it tries to read it. The rest of the representation is implementation-defined but will typically be something like the output just shown, including the name of the class and some distinguishing value such as the address of the object in memory. In Chapter 23 you'll see an example of how to define a method on **PRINT-OBJECT** to make objects of a certain class be printed in a more informative form.

Using the definition of `bank-account` just given, new objects will be created with their slots *unbound*. Any attempt to get the value of an unbound slot signals an error, so you must set a slot before you can read it.

```
(defparameter *account* (make-instance 'bank-account))  ==> *ACCOUNT*
(setf (slot-value *account* 'customer-name) "John Doe") ==> "John Doe"
(setf (slot-value *account* 'balance) 1000)             ==> 1000
```

Now you can access the value of the slots.

```
(slot-value *account* 'customer-name) ==> "John Doe"
(slot-value *account* 'balance)       ==> 1000
```

Object Initialization

Since you can't do much with an object with unbound slots, it'd be nice to be able to create objects with their slots already initialized. Common Lisp provides three ways to control the initial value of slots. The first two involve adding options to the slot specifier in the **DEFCLASS** form: with the `:initarg` option, you can specify a name that can then be used as a keyword parameter to **MAKE-INSTANCE** and whose argument will be stored in the slot. A second option, `:initform`, lets you specify a Lisp expression that will be used to compute a value for the slot if no `:initarg` argument is passed to **MAKE-INSTANCE**. Finally, for complete control over the initialization, you can define a method on the generic function **INITIALIZE-INSTANCE**, which is called by **MAKE-INSTANCE**.[5]

A slot specifier that includes options such as `:initarg` or `:initform` is written as a list starting with the name of the slot followed by the options. For example, if you want to modify the definition of `bank-account` to allow callers of **MAKE-INSTANCE** to pass the customer name and the initial balance and to provide a default value of zero dollars for the balance, you'd write this:

```
(defclass bank-account ()
  ((customer-name
    :initarg :customer-name)
   (balance
    :initarg :balance
    :initform 0)))
```

Now you can create an account and specify the slot values at the same time.

```
(defparameter *account*
  (make-instance 'bank-account :customer-name "John Doe" :balance 1000))

(slot-value *account* 'customer-name) ==> "John Doe"
(slot-value *account* 'balance)       ==> 1000
```

If you don't supply a `:balance` argument to **`MAKE-INSTANCE`**, the **`SLOT-VALUE`** of `balance` will be computed by evaluating the form specified with the `:initform` option. But if you don't supply a `:customer-name` argument, the `customer-name` slot will be unbound, and an attempt to read it before you set it will signal an error.

```
(slot-value (make-instance 'bank-account) 'balance)       ==> 0
(slot-value (make-instance 'bank-account) 'customer-name) ==> error
```

If you want to ensure that the customer name is supplied when the account is created, you can signal an error in the initform since it will be evaluated only if an initarg isn't supplied. You can also use initforms that generate a different value each time they're evaluated--the initform is evaluated anew for each object. To experiment with these techniques, you can modify the `customer-name` slot specifier and add a new slot, `account-number`, that's initialized with the value of an ever-increasing counter.

```
(defvar *account-numbers* 0)

(defclass bank-account ()
  ((customer-name
    :initarg :customer-name
    :initform (error "Must supply a customer name."))
   (balance
    :initarg :balance
    :initform 0)
   (account-number
    :initform (incf *account-numbers*))))
```

Most of the time the combination of `:initarg` and `:initform` options will be sufficient to properly initialize an object. However, while an initform can be any Lisp expression, it has no access to the object being initialized, so it can't initialize one slot based on the value of

another. For that you need to define a method on the generic function **INITIALIZE-INSTANCE**.

The primary method on **INITIALIZE-INSTANCE** specialized on **STANDARD-OBJECT** takes care of initializing slots based on their `:initarg` and `:initform` options. Since you don't want to disturb that, the most common way to add custom initialization code is to define an `:after` method specialized on your class.[6] For instance, suppose you want to add a slot `account-type` that needs to be set to one of the values `:gold`, `:silver`, or `:bronze` based on the account's initial balance. You might change your class definition to this, adding the `account-type` slot with no options:

```
(defclass bank-account ()
  ((customer-name
    :initarg :customer-name
    :initform (error "Must supply a customer name."))
   (balance
    :initarg :balance
    :initform 0)
   (account-number
    :initform (incf *account-numbers*))
   account-type))
```

Then you can define an `:after` method on **INITIALIZE-INSTANCE** that sets the `account-type` slot based on the value that has been stored in the `balance` slot.[7]

```
(defmethod initialize-instance :after ((account bank-account) &key)
  (let ((balance (slot-value account 'balance)))
    (setf (slot-value account 'account-type)
          (cond
            ((>= balance 100000) :gold)
            ((>= balance 50000) :silver)
            (t :bronze)))))
```

The **&key** in the parameter list is required to keep the method's parameter list congruent with the generic function's--the parameter list specified for the **INITIALIZE-INSTANCE** generic function includes

&key in order to allow individual methods to supply their own keyword parameters but doesn't require any particular ones. Thus, every method must specify **&key** even if it doesn't specify any **&key** parameters.

On the other hand, if an **INITIALIZE-INSTANCE** method specialized on a particular class does specify a **&key** parameter, that parameter becomes a legal parameter to **MAKE-INSTANCE** when creating an instance of that class. For instance, if the bank sometimes pays a percentage of the initial balance as a bonus when an account is opened, you could implement that using a method on **INITIALIZE-INSTANCE** that takes a keyword argument to specify the percentage of the bonus like this:

```
(defmethod initialize-instance :after ((account bank-account)
                                       &key opening-bonus-percentage)
  (when opening-bonus-percentage
    (incf (slot-value account 'balance)
          (* (slot-value account 'balance) (/ opening-bonus-percentage 100)))))
```

By defining this **INITIALIZE-INSTANCE** method, you make `:opening-bonus-percentage` a legal argument to **MAKE-INSTANCE** when creating a `bank-account` object.

```
CL-USER> (defparameter *acct* (make-instance
                                'bank-account
                                 :customer-name "Sally Sue"
                                 :balance 1000
                                 :opening-bonus-percentage 5))
*ACCT*
CL-USER> (slot-value *acct* 'balance)
1050
```

Accessor Functions

Between **MAKE-INSTANCE** and **SLOT-VALUE**, you have all the tools you need for creating and manipulating instances of your classes. Everything else you might want to do can be implemented in terms of those two functions. However, as anyone familiar with the principles of good object-oriented programming practices knows, directly accessing the slots (or fields or member variables) of an object can lead to fragile code. The problem is that directly accessing slots ties your code too tightly to the concrete structure of your class. For example, suppose you decide to change the definition of `bank-account` so that, instead of storing the current balance as a number, you store a list of time-stamped withdrawals and deposits. Code that directly accesses the `balance` slot will likely break if you change the class definition to remove the slot or to store the new list in the old slot. On the other hand, if you define a function, `balance`, that accesses the slot, you can redefine it later to preserve its behavior even if the internal representation changes. And code that uses such a function will continue to work without modification.

Another advantage to using accessor functions rather than direct access to slots via **SLOT-VALUE** is that they let you limit the ways outside code can modify a slot.[8] It may be fine for users of the `bank-account` class to get the current balance, but you may want all modifications to the balance to go through other functions you'll provide, such as `deposit` and `withdraw`. If clients know they're supposed to manipulate objects only through the published

functional API, you can provide a `balance` function but not make it **SETF**able if you want the balance to be read-only.

Finally, using accessor functions makes your code tidier since it helps you avoid lots of uses of the rather verbose **SLOT-VALUE** function.

It's trivial to define a function that reads the value of the `balance` slot.

```
(defun balance (account)
  (slot-value account 'balance))
```

However, if you know you're going to define subclasses of `bank-account`, it might be a good idea to define `balance` as a generic function. That way, you can provide different methods on `balance` for those subclasses or extend its definition with auxiliary methods. So you might write this instead:

```
(defgeneric balance (account))

(defmethod balance ((account bank-account))
  (slot-value account 'balance))
```

As I just discussed, you don't want callers to be able to directly set the balance, but for other slots, such as `customer-name`, you may also want to provide a function to set them. The cleanest way to define such a function is as a **SETF** function.

A **SETF** function is a way to extend **SETF**, defining a new kind of place that it knows how to set. The name of a **SETF** function is a two-item list whose first element is the symbol `setf` and whose second element is a symbol, typically the name of a function used to access the place the **SETF** function will set. A **SETF** function can take any number of arguments, but the first argument is always the value to

be assigned to the place.[9] You could, for instance, define a **SETF** function to set the `customer-name` slot in a `bank-account` like this:

```
(defun (setf customer-name) (name account)
  (setf (slot-value account 'customer-name) name))
```

After evaluating that definition, an expression like the following one:

```
(setf (customer-name my-account) "Sally Sue")
```

will be compiled as a call to the **SETF** function you just defined with "Sally Sue" as the first argument and the value of `my-account` as the second argument.

Of course, as with reader functions, you'll probably want your **SETF** function to be generic, so you'd actually define it like this:

```
(defgeneric (setf customer-name) (value account))

(defmethod (setf customer-name) (value (account bank-account))
  (setf (slot-value account 'customer-name) value))
```

And of course you'll also want to define a reader function for `customer-name`.

```
(defgeneric customer-name (account))

(defmethod customer-name ((account bank-account))
  (slot-value account 'customer-name))
```

This allows you to write the following:

```
(setf (customer-name *account*) "Sally Sue") ==> "Sally Sue"

(customer-name *account*)                    ==> "Sally Sue"
```

There's nothing hard about writing these accessor functions, but it wouldn't be in keeping with The Lisp Way to have to write them all by

hand. Thus, **DEFCLASS** supports three slot options that allow you to automatically create reader and writer functions for a specific slot.

The `:reader` option specifies a name to be used as the name of a generic function that accepts an object as its single argument. When the **DEFCLASS** is evaluated, the generic function is created, if it doesn't already exist. Then a method specializing its single argument on the new class and returning the value of the slot is added to the generic function. The name can be anything, but it's typical to name it the same as the slot itself. Thus, instead of explicitly writing the `balance` generic function and method as shown previously, you could change the slot specifier for the `balance` slot in the definition of `bank-account` to this:

```
(balance
 :initarg :balance
 :initform 0
 :reader balance)
```

The `:writer` option is used to create a generic function and method for setting the value of a slot. The function and method created follow the requirements for a **SETF** function, taking the new value as the first argument and returning it as the result, so you can define a **SETF** function by providing a name such as `(setf customer-name)`. For instance, you could provide reader and writer methods for `customer-name` equivalent to the ones you just wrote by changing the slot specifier to this:

```
(customer-name
 :initarg :customer-name
 :initform (error "Must supply a customer name.")
 :reader customer-name
 :writer (setf customer-name))
```

Since it's quite common to want both reader and writer functions, **DEFCLASS** also provides an option, `:accessor`, that creates both a reader function and the corresponding **SETF** function. So instead of the slot specifier just shown, you'd typically write this:

```
(customer-name
 :initarg :customer-name
 :initform (error "Must supply a customer name.")
 :accessor customer-name)
```

Finally, one last slot option you should know about is the `:documentation` option, which you can use to provide a string that documents the purpose of the slot. Putting it all together and adding a reader method for the `account-number` and `account-type` slots, the **DEFCLASS** form for the `bank-account` class would look like this:

```
(defclass bank-account ()
  ((customer-name
    :initarg :customer-name
    :initform (error "Must supply a customer name.")
    :accessor customer-name
    :documentation "Customer's name")
   (balance
    :initarg :balance
    :initform 0
    :reader balance
    :documentation "Current account balance")
   (account-number
    :initform (incf *account-numbers*)
    :reader account-number
    :documentation "Account number, unique within a bank.")
   (account-type
    :reader account-type
    :documentation "Type of account, one of :gold, :silver, or :bronze.")))
```

WITH-SLOTS and WITH-ACCESSORS

While using accessor functions will make your code easier to maintain, they can still be a bit verbose. And there will be times, when writing methods that implement the low-level behaviors of a class, that you may specifically want to access slots directly to set a slot that has no writer function or to get at the slot value without causing any auxiliary methods defined on the reader function to run.

This is what **SLOT-VALUE** is for; however, it's still quite verbose. To make matters worse, a function or method that accesses the same slot several times can become clogged with calls to accessor functions and **SLOT-VALUE**. For example, even a fairly simple method such as the following, which assesses a penalty on a `bank-account` if its balance falls below a certain minimum, is cluttered with calls to `balance` and **SLOT-VALUE**:

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (when (< (balance account) *minimum-balance*)
    (decf (slot-value account 'balance) (* (balance account) .01))))
```

And if you decide you want to directly access the slot value in order to avoid running auxiliary methods, it gets even more cluttered.

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (when (< (slot-value account 'balance) *minimum-balance*)
    (decf (slot-value account 'balance) (* (slot-value account 'balance) .01))))
```

Two standard macros, **WITH-SLOTS** and **WITH-ACCESSORS**, can help tidy up this clutter. Both macros create a block of code in which simple variable names can be used to refer to slots on a particular object. **WITH-SLOTS** provides direct access to the slots, as if by **SLOT-VALUE**, while **WITH-ACCESSORS** provides a shorthand for accessor methods.

The basic form of **WITH-SLOTS** is as follows:

```
(with-slots (slot*) instance-form
  body-form*)
```

Each element of *slots* can be either the name of a slot, which is also used as a variable name, or a two-item list where the first item is a name to use as a variable and the second is the name of the slot. The *instance-form* is evaluated once to produce the object whose slots will be accessed. Within the body, each occurrence of one of the variable names is translated to a call to **SLOT-VALUE** with the object and the appropriate slot name as arguments.[10] Thus, you can write `assess-low-balance-penalty` like this:

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (with-slots (balance) account
    (when (< balance *minimum-balance*)
      (decf balance (* balance .01)))))
```

or, using the two-item list form, like this:

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (with-slots ((bal balance)) account
    (when (< bal *minimum-balance*)
      (decf bal (* bal .01)))))
```

If you had defined `balance` with an `:accessor` rather than just a `:reader`, then you could also use **WITH-ACCESSORS**. The form of **WITH-ACCESSORS** is the same as **WITH-SLOTS** except each element of the slot list is a two-item list containing a variable name and the name of an accessor function. Within the body of **WITH-ACCESSORS**, a reference to one of the variables is equivalent to a call to the corresponding accessor function. If the accessor function is **SETF**able, then so is the variable.

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (with-accessors ((balance balance)) account
```

```
    (when (< balance *minimum-balance*)
      (decf balance (* balance .01)))))
```

The first `balance` is the name of the variable, and the second is the name of the accessor function; they don't have to be the same. You could, for instance, write a method to merge two accounts using two calls to **WITH-ACCESSORS**, one for each account.

```
(defmethod merge-accounts ((account1 bank-account) (account2 bank-account))
  (with-accessors ((balance1 balance)) account1
    (with-accessors ((balance2 balance)) account2
      (incf balance1 balance2)
      (setf balance2 0))))
```

The choice of whether to use **WITH-SLOTS** versus **WITH-ACCESSORS** is the same as the choice between **SLOT-VALUE** and an accessor function: low-level code that provides the basic functionality of a class may use **SLOT-VALUE** or **WITH-SLOTS** to directly manipulate slots in ways not supported by accessor functions or to explicitly avoid the effects of auxiliary methods that may have been defined on the accessor functions. But you should generally use accessor functions or **WITH-ACCESSORS** unless you have a specific reason not to.

Class-Allocated Slots

The last slot option you need to know about is `:allocation`. The value of `:allocation` can be either `:instance` or `:class` and defaults to `:instance` if not specified. When a slot has `:class` allocation, the slot has only a single value, which is stored in the class and shared by all instances.

However, `:class` slots are accessed the same as `:instance` slots--they're accessed with **`SLOT-VALUE`** or an accessor function, which means you can access the slot value only through an instance of the class even though it isn't actually stored in the instance. The `:initform` and `:initarg` options have essentially the same effect except the initform is evaluated once when the class is defined rather than each time an instance is created. On the other hand, passing an initarg to **`MAKE-INSTANCE`** will set the value, affecting all instances of the class.

Because you can't get at a class-allocated slot without an instance of the class, class-allocated slots aren't really equivalent to *static* or *class* fields in languages such as Java, C++, and Python.[11] Rather, class-allocated slots are used primarily to save space; if you're going to create many instances of a class and all instances are going to have a reference to the same object--say, a pool of shared resources--you can save the cost of each instance having its own reference by making the slot class-allocated.

Slots and Inheritance

As I discussed in the previous chapter, classes inherit behavior from their superclasses thanks to the generic function machinery--a method specialized on class `A` is applicable not only to direct instances of `A` but also to instances of `A`'s subclasses. Classes also inherit slots from their superclasses, but the mechanism is slightly different.

In Common Lisp a given object can have only one slot with a particular name. However, it's possible that more than one class in the inheritance hierarchy of a given class will specify a slot with a particular name. This can happen either because a subclass includes a slot specifier with the same name as a slot specified in a superclass or because multiple superclasses specify slots with the same name.

Common Lisp resolves these situations by merging all the specifiers with the same name from the new class and all its superclasses to create a single specifier for each unique slot name. When merging specifiers, different slot options are treated differently. For instance, since a slot can have only a single default value, if multiple classes specify an `:initform`, the new class uses the one from the most specific class. This allows a subclass to specify a different default value than the one it would otherwise inherit.

On the other hand, `:initarg`s needn't be exclusive--each `:initarg` option in a slot specifier creates a keyword parameter that can be used to initialize the slot; multiple parameters don't create a conflict,

so the new slot specifier contains all the `:initarg`s. Callers of **MAKE-INSTANCE** can use any of the `:initarg`s to initialize the slot. If a caller passes multiple keyword arguments that initialize the same slot, then the leftmost argument in the call to **MAKE-INSTANCE** is used.

Inherited `:reader`, `:writer`, and `:accessor` options aren't included in the merged slot specifier since the methods created by the superclass's **DEFCLASS** will already apply to the new class. The new class can, however, create its own accessor functions by supplying its own `:reader`, `:writer`, or `:accessor` options.

Finally, the `:allocation` option is, like `:initform`, determined by the most specific class that specifies the slot. Thus, it's possible for all instances of one class to share a `:class` slot while instances of a subclass may each have their own `:instance` slot of the same name. And a sub-subclass may then redefine it back to `:class` slot, so all instances of *that* class will again share a single slot. In the latter case, the slot shared by instances of the sub-subclass is different than the slot shared by the original superclass.

For instance, suppose you have these classes:

```
(defclass foo ()
  ((a :initarg :a :initform "A" :accessor a)
   (b :initarg :b :initform "B" :accessor b)))

(defclass bar (foo)
  ((a :initform (error "Must supply a value for a"))
   (b :initarg :the-b :accessor the-b :allocation :class)))
```

When instantiating the class `bar`, you can use the inherited initarg, `:a`, to specify a value for the slot `a` and, in fact, must do so to avoid an error, since the `:initform` supplied by `bar` supersedes the one inherited from `foo`. To initialize the `b` slot, you can use either the

inherited initarg `:b` or the new initarg `:the-b`. However, because of the `:allocation` option on the `b` slot in `bar`, the value specified will be stored in the slot shared by all instances of `bar`. That same slot can be accessed either with the method on the generic function `b` that specializes on `foo` or with the new method on the generic function `the-b` that specializes directly on `bar`. To access the `a` slot on either a `foo` or a `bar`, you'll continue to use the generic function `a`.

Usually merging slot definitions works quite nicely. However, it's important to be aware when using multiple inheritance that two unrelated slots that happen to have the same name can be merged into a single slot in the new class. Thus, methods specialized on different classes could end up manipulating the same slot when applied to a class that extends those classes. This isn't much of a problem in practice since, as you'll see in Chapter 21, you can use the package system to avoid collisions between names in independently developed pieces of code.

Multiple Inheritance

All the classes you've seen so far have had only a single direct superclass. Common Lisp also supports multiple inheritance--a class can have multiple direct superclasses, inheriting applicable methods and slot specifiers from all of them.

Multiple inheritance doesn't dramatically change any of the mechanisms of inheritance I've discussed so far--every user-defined class already has multiple superclasses since they all extend `STANDARD-OBJECT`, which extends `T`, and so have at least two superclasses. The wrinkle that multiple inheritance adds is that a class can have more than one *direct* superclass. This complicates the notion of class specificity that's used both when building the effective methods for a generic function and when merging inherited slot specifiers.

That is, if classes could have only a single direct superclass, ordering classes by specificity would be trivial--a class and all its superclasses could be ordered in a straight line starting from the class itself, followed by its single direct superclass, followed by *its* direct superclass, all the way up to `T`. But when a class has multiple direct superclasses, those superclasses are typically not related to each other--indeed, if one was a subclass of another, you wouldn't need to subclass both directly. In that case, the rule that subclasses are more specific than their superclasses isn't enough to order all the superclasses. So Common Lisp uses a second rule that sorts unrelated superclasses according to the order they're listed in the `DEFCLASS`'s direct superclass list--classes earlier in the list are

considered more specific than classes later in the list. This rule is admittedly somewhat arbitrary but does allow every class to have a linear *class precedence list*, which can be used to determine which superclasses should be considered more specific than others. Note, however, there's no global ordering of classes--each class has its own class precedence list, and the same classes can appear in different orders in different classes' class precedence lists.

To see how this works, let's add a class to the banking app: `money-market-account`. A money market account combines the characteristics of a checking account and a savings account: a customer can write checks against it, but it also earns interest. You might define it like this:

```
(defclass money-market-account (checking-account savings-account) ())
```

The class precedence list for `money-market-account` will be as follows:

```
(money-market-account
 checking-account
 savings-account
 bank-account
 standard-object
 t)
```

Note how this list satisfies both rules: every class appears before all its superclasses, and `checking-account` and `savings-account` appear in the order specified in **DEFCLASS**.

This class defines no slots of its own but will inherit slots from both of its direct superclasses, including the slots they inherit from their superclasses. Likewise, any method that's applicable to any class in the class precedence list will be applicable to a

`money-market-account` object. Because all slot specifiers for the same slot are merged, it doesn't matter that `money-market-account` inherits the same slot specifiers from `bank-account` twice. [12]

Multiple inheritance is easiest to understand when the different superclasses provide completely independent slots and behaviors. For instance, `money-market-account` will inherit slots and behaviors for dealing with checks from `checking-account` and slots and behaviors for computing interest from `savings-account`. You don't have to worry about the class precedence list for methods and slots inherited from only one superclass or another.

However, it's also possible to inherit different methods for the same generic function from different superclasses. In that case, the class precedence list does come into play. For instance, suppose the banking application defined a generic function `print-statement` used to generate monthly statements. Presumably there would already be methods for `print-statement` specialized on both `checking-account` and `savings-account`. Both of these methods will be applicable to instances of `money-market-account`, but the one specialized on `checking-account` will be considered more specific than the one on `savings-account` because `checking-account` precedes `savings-account` in `money-market-account`'s class precedence list.

Assuming the inherited methods are all primary methods and you haven't defined any other methods, the method specialized on `checking-account` will be used if you invoke `print-statement` on `money-market-account`. However, that won't necessarily give you the behavior you want since you probably want a money market

account's statement to contain elements of both a checking account and a savings account statement.

You can modify the behavior of `print-statement` for `money-market-account`s in a couple ways. One straightforward way is to define a new primary method specialized on `money-market-account`. This gives you the most control over the new behavior but will probably require more new code than some other options I'll discuss in a moment. The problem is that while you can use **`CALL-NEXT-METHOD`** to call "up" to the next most specific method, namely, the one specialized on `checking-account`, there's no way to invoke a particular less-specific method, such as the one specialized on `savings-account`. Thus, if you want to be able to reuse the code that prints the `savings-account` part of the statement, you'll need to break that code into a separate function, which you can then call directly from both the `money-market-account` and `savings-account` `print-statement` methods.

Another possibility is to write the primary methods of all three classes to call **`CALL-NEXT-METHOD`**. Then the method specialized on `money-market-account` will use **`CALL-NEXT-METHOD`** to invoke the method specialized on `checking-account`. When that method calls **`CALL-NEXT-METHOD`**, it will result in running the `savings-account` method since it will be the next most specific method according to `money-market-account`'s class precedence list.

Of course, if you're going to rely on a coding convention--that every method calls **`CALL-NEXT-METHOD`**--to ensure all the applicable methods run at some point, you should think about using auxiliary methods

instead. In this case, instead of defining primary methods on `print-statement` for `checking-account` and `savings-account`, you can define those methods as `:after` methods, defining a single primary method on `bank-account`. Then, `print-statement`, called on a `money-market-account`, will print a basic account statement, output by the primary method specialized on `bank-account`, followed by details output by the `:after` methods specialized on `savings-account` and `checking-account`. And if you want to add details specific to `money-market-account`s, you can define an `:after` method specialized on `money-market-account`, which will run last of all.

The advantage of using auxiliary methods is that it makes it quite clear which methods are primarily responsible for implementing the generic function and which ones are only contributing additional bits of functionality. The disadvantage is that you don't get fine-grained control over the order in which the auxiliary methods run--if you wanted the `checking-account` part of the statement to print before the `savings-account` part, you'd have to change the order in which the `money-market-account` subclasses those classes. But that's a fairly dramatic change that could affect other methods and inherited slots. In general, if you find yourself twiddling the order of the direct superclass list as a way of fine-tuning the behavior of specific methods, you probably need to step back and rethink your approach.

On the other hand, if you don't care exactly what the order is but want it to be consistent across several generic functions, then using auxiliary methods may be just the thing. For example, if in addition to `print-statement` you have a `print-detailed-statement` generic function, you can implement both functions using `:after` methods on

the various subclasses of `bank-account`, and the order of the parts of both a regular and a detailed statement will be the same.

Good Object-Oriented Design

That's about it for the main features of Common Lisp's object system. If you have lots of experience with object-oriented programming, you can probably see how Common Lisp's features can be used to implement good object-oriented designs. However, if you have less experience with object orientation, you may need to spend some time absorbing the object-oriented way of thinking. Unfortunately, that's a fairly large topic and beyond the scope of this book. Or, as the man page for Perl's object system puts it, "Now you need just to go off and buy a book about object-oriented design methodology and bang your forehead with it for the next six months or so." Or you can wait for some of the practical chapters, later in this book, where you'll see several examples of how these features are used in practice. For now, however, you're ready to take a break from all this theory of object orientation and turn to the rather different topic of how to make good use of Common Lisp's powerful, but sometimes cryptic, `FORMAT` function.

---

[1]Defining new methods for an existing class may seem strange to folks used to statically typed languages such as C++ and Java in which all the methods of a class must be defined as part of the class definition. But programmers with experience in dynamically typed object-oriented languages such as Smalltalk and Objective C will find nothing strange about adding new behaviors to existing classes.

[2]In other object-oriented languages, slots might be called *fields*, *member variables*, or *attributes*.

[3]As when naming functions and variables, it's not quite true that you can use *any* symbol as a class name--you can't use names defined by the language standard. You'll see in Chapter 21 how to avoid such name conflicts.

[4]The argument to **MAKE-INSTANCE** can actually be either the name of the class or a class object returned by the function **CLASS-OF** or **FIND-CLASS**.

[5]Another way to affect the values of slots is with the `:default-initargs` option to **DEFCLASS**. This option is used to specify forms that will be evaluated to provide arguments for specific initialization parameters that aren't given a value in a particular call to **MAKE-INSTANCE**. You don't need to worry about `:default-initargs` for now.

[6]Adding an `:after` method to **INITIALIZE-INSTANCE** is the Common Lisp analog to defining a constructor in Java or C++ or an `__init__` method in Python.

[7]One mistake you might make until you get used to using auxiliary methods is to define a method on **INITIALIZE-INSTANCE** but without the `:after` qualifier. If you do that, you'll get a new primary method that shadows the default one. You can remove the unwanted primary method using the functions **REMOVE-METHOD** and **FIND-METHOD**. Certain development environments may provide a graphical user interface to do the same thing.

```
(remove-method #'initialize-instance
  (find-method #'initialize-instance () (list (find-class 'bank-account))))
```

[8]Of course, providing an accessor function doesn't really limit anything since other code can still use **SLOT-VALUE** to get at slots directly. Common Lisp doesn't provide strict encapsulation of slots the way some languages such as C++ and Java do; however, if the author of a class provides accessor functions and you ignore them, using **SLOT-VALUE** instead, you had better know what you're doing. It's also possible to use the package system, which I'll discuss in Chapter 21, to make it even more obvious that certain slots aren't to be accessed directly, by not exporting the names of the slots.

[9]One consequence of defining a **SETF** function--say, `(setf foo)`--is that if you also define the corresponding accessor function, `foo` in this case, you can use all the modify macros built upon **SETF**, such as **INCF**, **DECF**, **PUSH**, and **POP**, on the new kind of place.

[10]The "variable" names provided by **WITH-SLOTS** and **WITH-ACCESSORS** aren't true variables; they're implemented using a special kind of macro, called a *symbol macro*, that allows a simple name to expand into arbitrary code. Symbol macros were introduced into the language to support **WITH-SLOTS** and **WITH-ACCESSORS**, but you can also use them for your own purposes. I'll discuss them in a bit more detail in Chapter 20.

[11]The Meta Object Protocol (MOP), which isn't part of the language standard but is supported by most Common Lisp implementations, provides a function, `class-prototype`, that returns an instance of a class that can be used to access class slots. If you're using an

implementation that supports the MOP and happen to be translating some code from another language that makes heavy use of static or class fields, this may give you a way to ease the translation. But it's not all that idiomatic.

[12] In other words, Common Lisp doesn't suffer from the *diamond inheritance* problem the way, say, C++ does. In C++, when one class subclasses two classes that both inherit a member variable from a common superclass, the bottom class inherits the member variable twice, leading to no end of confusion.

Copyright © 2003-2005, Peter Seibel

18. A Few FORMAT Recipes

Common Lisp's **FORMAT** function is--along with the extended **LOOP** macro--one of the two Common Lisp features that inspires a strong emotional response in a lot of Common Lisp users. Some love it; others hate it.[1]

FORMAT's fans love it for its great power and concision, while its detractors hate it because of the potential for misuse and its opacity. Complex **FORMAT** control strings sometimes bear a suspicious resemblance to line noise, but **FORMAT** remains popular with Common Lispers who like to be able to generate little bits of human-readable output without having to clutter their code with lots of output-generating code. While **FORMAT**'s control strings can be cryptic, at least a single **FORMAT** expression doesn't clutter things up too badly. For instance, suppose you want to print the values in a list delimited with commas. You could write this:

```
(loop for cons on list
    do (format t "~a" (car cons))
    when (cdr cons) do (format t ", "))
```

That's not too bad, but anyone reading this code has to mentally parse it just to figure out that all it's doing is printing the contents of `list` to standard output. On the other hand, you can tell at a glance that the following expression is printing `list`, in some form, to standard output:

```
(format t "~{~a~^, ~}" list)
```

If you care exactly what form the output will take, then you'll have to examine the control string, but if all you want is a first-order approximation of what this line of code is doing, that's immediately available.

At any rate, you should have at least a reading knowledge of `FORMAT`, and it's worth getting a sense of what it can do before you affiliate yourself with the pro- or anti-`FORMAT` camp. It's also important to understand at least the basics of `FORMAT` because other standard functions, such as the condition-signaling functions discussed in the next chapter, use `FORMAT`-style control strings to generate output.

To further complicate matters, `FORMAT` supports three quite different kinds of formatting: printing tables of data, *pretty-printing* s-expressions, and generating human-readable messages with interpolated values. Printing tables of data as text is a bit passé these days; it's one of those reminders that Lisp is nearly as old as FORTRAN. In fact, several of the directives you can use to print floating-point values in fixed-width fields were based quite directly on FORTRAN *edit descriptors*, which are used in FORTRAN to read and print columns of data arranged in fixed-width fields. However, using Common Lisp as a FORTRAN replacement is beyond the scope of this book, so I won't discuss those aspects of `FORMAT`.

Pretty-printing is likewise beyond the scope of this book--not because it's passé but just because it's too big a topic. Briefly, the Common Lisp pretty printer is a customizable system for printing block-structured data such as--but not limited to--s-expressions while varying indentation and dynamically adding line breaks as needed.

It's a great thing when you need it, but it's not often needed in day-to-day programming.[2]

Instead, I'll focus on the parts of `FORMAT` you can use to generate human-readable strings with interpolated values. Even limiting the scope in that way, there's still a fair bit to cover. You shouldn't feel obliged to remember every detail described in this chapter. You can get quite far with just a few `FORMAT` idioms. I'll describe the most important features of `FORMAT` first; it's up to you how much of a `FORMAT` wizard you want to become.

The FORMAT Function

As you've seen in previous chapters, the `FORMAT` function takes two required arguments: a destination for its output and a control string that contains literal text and embedded *directives*. Any additional arguments provide the values used by the directives in the control string that interpolate values into the output. I'll refer to these arguments as *format arguments*.

The first argument to `FORMAT`, the destination for the output, can be `T`, `NIL`, a stream, or a string with a fill pointer. `T` is shorthand for the stream `*STANDARD-OUTPUT*`, while `NIL` causes `FORMAT` to generate its output to a string, which it then returns.[3] If the destination is a stream, the output is written to the stream. And if the destination is a string with a fill pointer, the formatted output is added to the end of the string and the fill pointer is adjusted appropriately. Except when the destination is `NIL` and it returns a string, `FORMAT` returns `NIL`.

The second argument, the control string, is, in essence, a program in the `FORMAT` language. The `FORMAT` language isn't Lispy at all--its basic syntax is based on characters, not s-expressions, and it's optimized for compactness rather than easy comprehension. This is why a complex `FORMAT` control string can end up looking like line noise.

Most of `FORMAT`'s directives simply interpolate an argument into the output in one form or another. Some directives, such as `~%`, which causes `FORMAT` to emit a newline, don't consume any arguments. And others, as you'll see, can consume more than one argument. One directive even allows you to jump around in the list of arguments in

order to process the same argument more than once or to skip certain arguments in certain situations. But before I discuss specific directives, let's look at the general syntax of a directive.

FORMAT Directives

All directives start with a tilde (`~`) and end with a single character that identifies the directive. You can write the character in either upper- or lowercase. Some directives take *prefix parameters*, which are written immediately following the tilde, separated by commas, and used to control things such as how many digits to print after the decimal point when printing a floating-point number. For example, the `~$` directive, one of the directives used to print floating-point values, by default prints two digits following the decimal point.

```
CL-USER> (format t "~$" pi)
3.14
NIL
```

However, with a prefix parameter, you can specify that it should print its argument to, say, five decimal places like this:

```
CL-USER> (format t "~5$" pi)
3.14159
NIL
```

The values of prefix parameters are either numbers, written in decimal, or characters, written as a single quote followed by the desired character. The value of a prefix parameter can also be derived from the format arguments in two ways: A prefix parameter of `v` causes **`FORMAT`** to consume one format argument and use its value for the prefix parameter. And a prefix parameter of `#` will be evaluated as the number of remaining format arguments. For example:

```
CL-USER> (format t "~v$" 3 pi)
3.142
NIL
CL-USER> (format t "~#$" pi)
```

```
3.1
NIL
```

I'll give some more realistic examples of how you can use the `#` argument in the section "Conditional Formatting."

You can also omit prefix parameters altogether. However, if you want to specify one parameter but not the ones before it, you must include a comma for each unspecified parameter. For instance, the `~F` directive, another directive for printing floating-point values, also takes a parameter to control the number of decimal places to print, but it's the second parameter rather than the first. If you want to use `~F` to print a number to five decimal places, you can write this:

```
CL-USER> (format t "~,5f" pi)
3.14159
NIL
```

You can also modify the behavior of some directives with colon and at-sign *modifiers*, which are placed after any prefix parameters and before the directive's identifying character. These modifiers change the behavior of the directive in small ways. For instance, with a colon modifier, the `~D` directive used to output integers in decimal emits the number with commas separating every three digits, while the at-sign modifier causes `~D` to include a plus sign when the number is positive.

```
CL-USER> (format t "~d" 1000000)
1000000
NIL
CL-USER> (format t "~:d" 1000000)
1,000,000
NIL
CL-USER> (format t "~@d" 1000000)
+1000000
NIL
```

When it makes sense, you can combine the colon and at-sign modifiers to get both modifications.

```
CL-USER> (format t "~:@d" 1000000)
+1,000,000
NIL
```

In directives where the two modified behaviors can't be meaningfully combined, using both modifiers is either undefined or given a third meaning.

Basic Formatting

Now you're ready to look at specific directives. I'll start with several of the most commonly used directives, including some you've seen in previous chapters.

The most general-purpose directive is `~A`, which consumes one format argument of any type and outputs it in *aesthetic* (human-readable) form. For example, strings are output without quotation marks or escape characters, and numbers are output in a natural way for the type of number. If you just want to emit a value for human consumption, this directive is your best bet.

```
(format nil "The value is: ~a" 10)           ==> "The value is: 10"
(format nil "The value is: ~a" "foo")        ==> "The value is: foo"
(format nil "The value is: ~a" (list 1 2 3)) ==> "The value is: (1 2 3)"
```

A closely related directive, `~S`, likewise consumes one format argument of any type and outputs it. However, `~S` tries to generate output that can be read back in with **`READ`**. Thus, strings will be enclosed in quotation marks, symbols will be package-qualified when necessary, and so on. Objects that don't have a **`READ`**able representation are printed with the unreadable object syntax, `#<>`. With a colon modifier, both the `~A` and `~S` directives emit **`NIL`** as `()` rather than **`NIL`**. Both the `~A` and `~S` directives also take up to four prefix parameters, which can be used to control whether padding is added after (or before with the at-sign modifier) the value, but those parameters are only really useful for generating tabular data.

The other two most frequently used directives are `~%`, which emits a newline, and `~&`, which emits a *fresh line*. The difference between the

two is that `~%` always emits a newline, while `~&` emits one only if it's not already at the beginning of a line. This is handy when writing loosely coupled functions that each generate a piece of output and that need to be combined in different ways. For instance, if one function generates output that ends with a newline (`~%`) and another function generates some output that starts with a fresh line (`~&`), you don't have to worry about getting an extra blank line if you call them one after the other. Both of these directives can take a single prefix parameter that specifies the number of newlines to emit. The `~%` directive will simply emit that many newline characters, while the `~&` directive will emit either *n* - 1 or *n* newlines, depending on whether it starts at the beginning of a line.

Less frequently used is the related `~~` directive, which causes **FORMAT** to emit a literal tilde. Like the `~%` and `~&` directives, it can be parameterized with a number that controls how many tildes to emit.

Character and Integer Directives

In addition to the general-purpose directives, `~A` and `~S`, **`FORMAT`** supports several directives that can be used to emit values of specific types in particular ways. One of the simplest of these is the `~C` directive, which is used to emit characters. It takes no prefix arguments but can be modified with the colon and at-sign modifiers. Unmodified, its behavior is no different from `~A` except that it works only with characters. The modified versions are more useful. With a colon modifier, `~:C` outputs *nonprinting* characters such as space, tab, and newline by name. This is useful if you want to emit a message to the user about some character. For instance, the following:

```
(format t "Syntax error. Unexpected character: ~:c" char)
```

can emit messages like this:

```
Syntax error. Unexpected character: a
```

but also like the following:

```
Syntax error. Unexpected character: Space
```

With the at-sign modifier, `~@C` will emit the character in Lisp's literal character syntax.

```
CL-USER> (format t "~@c~%" #\a)
#\a
NIL
```

With both the colon and at-sign modifiers, the `~C` directive can print extra information about how to enter the character at the keyboard if it requires special key combinations. For instance, on the Macintosh,

in certain applications you can enter a null character (character code 0 in ASCII or in any ASCII superset such as ISO-8859-1 or Unicode) by pressing the Control key and typing @. In OpenMCL, if you print the null character with the `~:C` directive, it tells you this:

```
(format nil "~:@c" (code-char 0)) ==> "^@ (Control @)"
```

However, not all Lisps implement this aspect of the `~C` directive. And even if they do, it may or may not be accurate--for instance, if you're running OpenMCL in SLIME, the `C-@` key chord is intercepted by Emacs, invoking `set-mark-command`.[4]

Format directives dedicated to emitting numbers are another important category. While you can use the `~A` and `~S` directives to emit numbers, if you want fine control over how they're printed, you need to use one of the number-specific directives. The numeric directives can be divided into two subcategories: directives for formatting integer values and directives for formatting floating-point values.

Five closely related directives format integer values: `~D`, `~X`, `~O`, `~B`, and `~R`. The most frequently used is the `~D` directive, which outputs integers in base 10.

```
(format nil "~d" 1000000) ==> "1000000"
```

As I mentioned previously, with a colon modifier it adds commas.

```
(format nil "~:d" 1000000) ==> "1,000,000"
```

And with an at-sign modifier, it always prints a sign.

```
(format nil "~@d" 1000000) ==> "+1000000"
```

And the two modifiers can be combined.

```
(format nil "~:@d" 1000000) ==> "+1,000,000"
```

The first prefix parameter can specify a minimum width for the output, and the second parameter can specify a padding character to use. The default padding character is space, and padding is always inserted before the number itself.

```
(format nil "~12d" 1000000)    ==> "     1000000"
(format nil "~12,'0d" 1000000) ==> "000001000000"
```

These parameters are handy for formatting things such as dates in a fixed-width format.

```
(format nil "~4,'0d-~2,'0d-~2,'0d" 2005 6 10) ==> "2005-06-10"
```

The third and fourth parameters are used in conjunction with the colon modifier: the third parameter specifies the character to use as the separator between groups and digits, and the fourth parameter specifies the number of digits per group. These parameters default to a comma and the number 3. Thus, you can use the directive `~:D` without parameters to output large integers in standard format for the United States but can change the comma to a period and the grouping from 3 to 4 with `~,,'.,4D`.

```
(format nil "~:d" 100000000)       ==> "100,000,000"
(format nil "~,,'.,4:d" 100000000) ==> "1.0000.0000"
```

Note that you must use commas to hold the places of the unspecified width and padding character parameters, allowing them to keep their default values.

The `~X`, `~O`, and `~B` directives work just like the `~D` directive except they emit numbers in hexadecimal (base 16), octal (base 8), and

binary (base 2).

```
(format nil "~x" 1000000) ==> "f4240"
(format nil "~o" 1000000) ==> "3641100"
(format nil "~b" 1000000) ==> "11110100001001000000"
```

Finally, the `~R` directive is the general *radix* directive. Its first parameter is a number between 2 and 36 (inclusive) that indicates what base to use. The remaining parameters are the same as the four parameters accepted by the `~D`, `~X`, `~O`, and `~B` directives, and the colon and at-sign modifiers modify its behavior in the same way. The `~R` directive also has some special behavior when used with no prefix parameters, which I'll discuss in the section "English-Language Directives."

Floating-Point Directives

Four directives format floating-point values: `~F`, `~E`, `~G`, and `~$`. The first three of these are the directives based on FORTRAN's edit descriptors. I'll skip most of the details of those directives since they mostly have to do with formatting floating-point values for use in tabular form. However, you can use the `~F`, `~E`, and `~$` directives to interpolate floating-point values into text. The `~G`, or *general,* floating-point directive, on the other hand, combines aspects of the `~F` and `~E` directives in a way that only really makes sense for generating tabular output.

The `~F` directive emits its argument, which should be a number,[5] in decimal format, possibly controlling the number of digits after the decimal point. The `~F` directive is, however, allowed to use computerized scientific notation if the number is sufficiently large or small. The `~E` directive, on the other hand, always emits numbers in computerized scientific notation. Both of these directives take a number of prefix parameters, but you need to worry only about the second, which controls the number of digits to print after the decimal point.

```
(format nil "~f" pi)   ==> "3.141592653589793d0"
(format nil "~,4f" pi) ==> "3.1416"
(format nil "~e" pi)   ==> "3.141592653589793d+0"
(format nil "~,4e" pi) ==> "3.1416d+0"
```

The `~$`, or monetary, directive is similar to `~F` but a bit simpler. As its name suggests, it's intended for emitting monetary units. With no parameters, it's basically equivalent to `~,2F`. To modify the number of digits printed after the decimal point, you use the *first* parameter,

while the second parameter controls the minimum number of digits to print before the decimal point.

```
(format nil "~$" pi)    ==> "3.14"
(format nil "~2,4$" pi) ==> "0003.14"
```

All three directives, `~F`, `~E`, and `~$`, can be made to always print a sign, plus or minus, with the at-sign modifier.[6]

English-Language Directives

Some of the handiest **FORMAT** directives for generating human-readable messages are the ones for emitting English text. These directives allow you to emit numbers as English words, to emit plural markers based on the value of a format argument, and to apply case conversions to sections of **FORMAT**'s output.

The `~R` directive, which I discussed in "Character and Integer Directives," when used with no base specified, prints numbers as English words or Roman numerals. When used with no prefix parameter and no modifiers, it emits the number in words as a cardinal number.

```
(format nil "~r" 1234) ==> "one thousand two hundred thirty-four"
```

With the colon modifier, it emits the number as an ordinal.

```
(format nil "~:r" 1234) ==> "one thousand two hundred thirty-fourth"
```

And with an at-sign modifier, it emits the number as a Roman numeral; with both an at-sign and a colon, it emits "old-style" Roman numerals in which fours and nines are written as IIII and VIIII instead of IV and IX.

```
(format nil "~@r" 1234)  ==> "MCCXXXIV"
(format nil "~:@r" 1234) ==> "MCCXXXIIII"
```

For numbers too large to be represented in the given form, `~R` behaves like `~D`.

To help you generate messages with words properly pluralized, **FORMAT** provides the `~P` directive, which simply emits an *s* unless the

corresponding argument is `1`.

```
(format nil "file~p" 1)  ==> "file"
(format nil "file~p" 10) ==> "files"
(format nil "file~p" 0)  ==> "files"
```

Typically, however, you'll use `~P` with the colon modifier, which causes it to reprocess the previous format argument.

```
(format nil "~r file~:p" 1)  ==> "one file"
(format nil "~r file~:p" 10) ==> "ten files"
(format nil "~r file~:p" 0)  ==> "zero files"
```

With the at-sign modifier, which can be combined with the colon modifier, `~P` emits either *y* or *ies*.

```
(format nil "~r famil~:@p" 1)  ==> "one family"
(format nil "~r famil~:@p" 10) ==> "ten families"
(format nil "~r famil~:@p" 0)  ==> "zero families"
```

Obviously, `~P` can't solve all pluralization problems and is no help for generating messages in other languages, but it's handy for the cases it does handle. And the `~[` directive, which I'll discuss in a moment, gives you a more flexible way to conditionalize parts of **FORMAT**'s output.

The last directive for dealing with emitting English text is `~(`, which allows you to control the case of text in the output. Each `~(` is paired with a `~)`, and all the output generated by the portion of the control string between the two markers will be converted to all lowercase.

```
(format nil "~(~a~)" "FOO") ==> "foo"
(format nil "~(~@r~)" 124)  ==> "cxxiv"
```

You can modify `~(` with an at sign to make it capitalize the first word in a section of text, with a colon to make it to capitalize all words, and with both modifiers to convert all text to uppercase. (A *word* for the

purpose of this directive is a sequence of alphanumeric characters delimited by nonalphanumeric characters or the ends of the text.)

```
(format nil "~(~a~)" "tHe Quick BROWN foX")   ==> "the quick brown fox"
(format nil "~@(~a~)" "tHe Quick BROWN foX")  ==> "The quick brown fox"
(format nil "~:(~a~)" "tHe Quick BROWN foX")  ==> "The Quick Brown Fox"
(format nil "~:@(~a~)" "tHe Quick BROWN foX") ==> "THE QUICK BROWN FOX"
```

Conditional Formatting

In addition to directives that interpolate arguments and modify other output, **FORMAT** provides several directives that implement simple control constructs within the control string. One of these, which you used in Chapter 9, is the *conditional* directive `~[`. This directive is closed by a corresponding `~]`, and in between are a number of clauses separated by `~;`. The job of the `~[` directive is to pick one of the clauses, which is then processed by **FORMAT**. With no modifiers or parameters, the clause is selected by numeric index; the `~[` directive consumes a format argument, which should be a number, and takes the *nth* (zero-based) clause where *N* is the value of the argument.

```
(format nil "~[cero~;uno~;dos~]" 0) ==> "cero"
(format nil "~[cero~;uno~;dos~]" 1) ==> "uno"
(format nil "~[cero~;uno~;dos~]" 2) ==> "dos"
```

If the value of the argument is greater than the number of clauses, nothing is printed.

```
(format nil "~[cero~;uno~;dos~]" 3) ==> ""
```

However, if the last clause separator is `~:;` instead of `~;`, then the last clause serves as a default clause.

```
(format nil "~[cero~;uno~;dos~:;mucho~]" 3)   ==> "mucho"
(format nil "~[cero~;uno~;dos~:;mucho~]" 100) ==> "mucho"
```

It's also possible to specify the clause to be selected using a prefix parameter. While it'd be silly to use a literal value in the control string, recall that `#` used as a prefix parameter means the number of arguments remaining to be processed. Thus, you can define a format string such as the following:

```
(defparameter *list-etc*
  "~#[NONE~;~a~;~a and ~a~:;~a, ~a~]~#[~; and ~a~:;, ~a, etc~].")
```

and then use it like this:

```
(format nil *list-etc*)                ==> "NONE."
(format nil *list-etc* 'a)             ==> "A."
(format nil *list-etc* 'a 'b)          ==> "A and B."
(format nil *list-etc* 'a 'b 'c)       ==> "A, B and C."
(format nil *list-etc* 'a 'b 'c 'd)    ==> "A, B, C, etc."
(format nil *list-etc* 'a 'b 'c 'd 'e) ==> "A, B, C, etc."
```

Note that the control string actually contains two `~[~]` directives--both of which use `#` to select the clause to use. The first consumes between zero and two arguments, while the second consumes one more, if available. **FORMAT** will silently ignore any arguments not consumed while processing the control string.

With a colon modifier, the `~[` can contain only two clauses; the directive consumes a single argument and processes the first clause if the argument is **NIL** and the second clause is otherwise. You used this variant of `~[` in Chapter 9 to generate pass/fail messages, like this:

```
(format t "~:[FAIL~;pass~]" test-result)
```

Note that either clause can be empty, but the directive must contain a `~;`.

Finally, with an at-sign modifier, the `~[` directive can have only one clause. The directive consumes one argument and, if it's non-**NIL**, processes the clause after backing up to make the argument available to be consumed again.

```
(format nil "~@[x = ~a ~]~@[y = ~a~]" 10 20)   ==> "x = 10 y = 20"
(format nil "~@[x = ~a ~]~@[y = ~a~]" 10 nil)  ==> "x = 10 "
(format nil "~@[x = ~a ~]~@[y = ~a~]" nil 20)  ==> "y = 20"
(format nil "~@[x = ~a ~]~@[y = ~a~]" nil nil) ==> ""
```

Iteration

Another **FORMAT** directive that you've seen already, in passing, is the iteration directive `~{`. This directive tells **FORMAT** to iterate over the elements of a list or over the implicit list of the format arguments.

With no modifiers, `~{` consumes one format argument, which must be a list. Like the `~[` directive, which is always paired with a `~]` directive, the `~{` directive is always paired with a closing `~}`. The text between the two markers is processed as a control string, which draws its arguments from the list consumed by the `~{` directive. **FORMAT** will repeatedly process this control string for as long as the list being iterated over has elements left. In the following example, the `~{` consumes the single format argument, the list `(1 2 3)`, and then processes the control string `"~a, "`, repeating until all the elements of the list have been consumed.

```
(format nil "~{~a, ~}" (list 1 2 3)) ==> "1, 2, 3, "
```

However, it's annoying that in the output the last element of the list is followed by a comma and a space. You can fix that with the `~^` directive; within the body of a `~{` directive, the `~^` causes the iteration to stop immediately, without processing the rest of the control string, when no elements remain in the list. Thus, to avoid printing the comma and space after the last element of a list, you can precede them with a `~^`.

```
(format nil "~{~a~^, ~}" (list 1 2 3)) ==> "1, 2, 3"
```

The first two times through the iteration, there are still unprocessed elements in the list when the `~^` is processed. The third time through,

however, after the `~a` directive consumes the `3`, the `~^` will cause **FORMAT** to break out of the iteration without printing the comma and space.

With an at-sign modifier, `~{` processes the remaining format arguments as a list.

```
(format nil "~@{~a~^, ~}" 1 2 3) ==> "1, 2, 3"
```

Within the body of a `~{...~}`, the special prefix parameter `#` refers to the number of items remaining to be processed in the list rather than the number of remaining format arguments. You can use that, along with the `~[` directive, to print a comma-separated list with an "and" before the last item like this:

```
(format nil "~{~a~#[~;, and ~:;, ~]~}" (list 1 2 3)) ==> "1, 2, and 3"
```

However, that doesn't really work right if the list is two items long because it adds an extra comma.

```
(format nil "~{~a~#[~;, and ~:;, ~]~}" (list 1 2)) ==> "1, and 2"
```

You could fix that in a bunch of ways. The following takes advantage of the behavior of `~@{` when nested inside another `~{` or `~@{` directive--it iterates over whatever items remain in the list being iterated over by the outer `~{`. You can combine that with a `~#[` directive to make the following control string for formatting lists according to English grammar:

```
(defparameter *english-list*
  "~{~#[~;~a~;~a and ~a~:;~@{~a~#[~;, and ~:;, ~]~}~]~}")

(format nil *english-list* '())        ==> ""
(format nil *english-list* '(1))       ==> "1"
(format nil *english-list* '(1 2))     ==> "1 and 2"
(format nil *english-list* '(1 2 3))   ==> "1, 2, and 3"
(format nil *english-list* '(1 2 3 4)) ==> "1, 2, 3, and 4"
```

While that control string verges on being "write-only" code, it's not too hard to understand if you take it a bit at a time. The outer `~{...~}` will consume and iterate over a list. The whole body of the iteration then consists of a `~#[...~]`; the output generated each time through the iteration will thus depend on the number of items left to be processed from the list. Splitting apart the `~#[...~]` directive on the `~;` clause separators, you can see that it's made up of four clauses, the last of which is a default clause because it's preceded by a `~:;` rather than a plain `~;`. The first clause, for when there are zero elements to be processed, is empty, which makes sense--if there are no more elements to be processed, the iteration would've stopped already. The second clause handles the case of one element with a simple `~a` directive. Two elements are handled with `"~a and ~a"`. And the default clause, which handles three or more elements, consists of another iteration directive, this time using `~@{` to iterate over the remaining elements of the list being processed by the outer `~{`. And the body of that iteration is the control string that can handle a list of three or more elements correctly, which is fine in this context. Because the `~@{` loop consumes all the remaining list items, the outer loop iterates only once.

If you wanted to print something special such as "<empty>" when the list was empty, you have a couple ways to do it. Perhaps the easiest is to put the text you want into the first (zeroth) clause of the outer `~#[` and then add a colon modifier to the closing `~}` of the outer iteration--the colon forces the iteration to be run at least once, even if the list is empty, at which point **FORMAT** processes the zeroth clause of the conditional directive.

```
(defparameter *english-list*
  "~{~#[<empty>~;~a~;~a and ~a~:;~@{~a~#[~;, and ~:;, ~]~}~]~:}")

(format nil *english-list* '()) ==> "<empty>"
```

Amazingly, the `~{` directive provides even more variations with different combinations of prefix parameters and modifiers. I won't discuss them other than to say you can use an integer prefix parameter to limit the maximum number of iterations and that, with a colon modifier, each element of the list (either an actual list or the list constructed by the `~@{` directive) must itself be a list whose elements will then be used as arguments to the control string in the `~:{...~}` directive.

Hop, Skip, Jump

A much simpler directive is the `~*` directive, which allows you to jump around in the list of format arguments. In its basic form, without modifiers, it simply skips the next argument, consuming it without emitting anything. More often, however, it's used with a colon modifier, which causes it to move backward, allowing the same argument to be used a second time. For instance, you can use `~:*` to print a numeric argument once as a word and once in numerals like this:

```
(format nil "~r ~:*(~d)" 1) ==> "one (1)"
```

Or you could implement a directive similar to `~:P` for an irregular plural by combing `~:*` with `~[`.

```
(format nil "I saw ~r el~:*~[ves~;f~:;ves~]." 0) ==> "I saw zero elves."
(format nil "I saw ~r el~:*~[ves~;f~:;ves~]." 1) ==> "I saw one elf."
(format nil "I saw ~r el~:*~[ves~;f~:;ves~]." 2) ==> "I saw two elves."
```

In this control string, the `~R` prints the format argument as a cardinal number. Then the `~:*` directive backs up so the number is also used as the argument to the `~[` directive, selecting between the clauses for when the number is zero, one, or anything else.[7]

Within an `~{` directive, `~*` skips or backs up over the items in the list. For instance, you could print only the keys of a plist like this:

```
(format nil "~{~s~*~^ ~}" '(:a 10 :b 20)) ==> ":A :B"
```

The `~*` directive can also be given a prefix parameter. With no modifiers or with the colon modifier, this parameter specifies the number of arguments to move forward or backward and defaults to

one. With an at-sign modifier, the prefix parameter specifies an absolute, zero-based index of the argument to jump to, defaulting to zero. The at-sign variant of `~*` can be useful if you want to use different control strings to generate different messages for the same arguments and if different messages need to use the arguments in different orders.[8]

And More . . .

And there's more--I haven't mentioned the `~?` directive, which can take snippets of control strings from the format arguments or the `~/` directive, which allows you to call an arbitrary function to handle the next format argument. And then there are all the directives for generating tabular and pretty-printed output. But the directives discussed in this chapter should be plenty for the time being.

In the next chapter, you'll move onto Common Lisp's condition system, the Common Lisp analog to other languages' exception and error handling systems.

---

[1]Of course, most folks realize it's not worth getting that worked up over *anything* in a programming language and use it or not without a lot of angst. On the other hand, it's interesting that these two features are the two features in Common Lisp that implement what are essentially domain-specific languages using a syntax not based on s-expressions. The syntax of `FORMAT`'s control strings is character based, while the extended `LOOP` macro can be understood only in terms of the grammar of the `LOOP` keywords. That one of the common knocks on both `FORMAT` and `LOOP` is that they "aren't Lispy enough" is evidence that Lispers really do like the s-expression syntax.

[2]Readers interested in the pretty printer may want to read the paper "XP: A Common Lisp Pretty Printing System" by Richard Waters. It's a description of the pretty printer that was eventually incorporated into Common Lisp. You can download it from `ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-1102a.pdf`.

[3]To slightly confuse matters, most other I/O functions also accept `T` and `NIL` as *stream designators* but with a different meaning: as a stream designator, `T` designates the bidirectional stream `*TERMINAL-IO*`, while `NIL` designates `*STANDARD-OUTPUT*` as an output stream and `*STANDARD-INPUT*` as an input stream.

[4]This variant on the `~C` directive makes more sense on platforms like the Lisp Machines where key press events were represented by Lisp characters.

[5]Technically, if the argument isn't a real number, `~F` is supposed to format it as if by the `~D` directive, which in turn behaves like the `~A` directive if the argument isn't a number, but not all implementations get this right.

[6]Well, that's what the language standard says. For some reason, perhaps rooted in a common ancestral code base, several Common Lisp implementations don't implement this aspect of the `~F` directive correctly.

[7]If you find "I saw zero elves" to be a bit clunky, you could use a slightly more elaborate format string that makes another use of `~:*` like this:

```
(format nil "I saw ~[no~:;~:*~r~] el~:*~[ves~;f~:;ves~]." 0) ==> "I saw no elves."
(format nil "I saw ~[no~:;~:*~r~] el~:*~[ves~;f~:;ves~]." 1) ==> "I saw one elf."
(format nil "I saw ~[no~:;~:*~r~] el~:*~[ves~;f~:;ves~]." 2) ==> "I saw two elves."
```

[8]This kind of problem can arise when trying to localize an application and translate human-readable messages into different languages. **FORMAT** can help with some of these problems but is by no means a full-blown localization system.

Copyright Š 2003-2005, Peter Seibel

19. Beyond Exception Handling: Conditions and Restarts

One of Lisp's great features is its *condition* system. It serves a similar purpose to the exception handling systems in Java, Python, and C++ but is more flexible. In fact, its flexibility extends beyond error handling--conditions are more general than exceptions in that a condition can represent any occurrence during a program's execution that may be of interest to code at different levels on the call stack. For example, in the section "Other Uses for Conditions," you'll see that conditions can be used to emit warnings without disrupting execution of the code that emits the warning while allowing code higher on the call stack to control whether the warning message is printed. For the time being, however, I'll focus on error handling.

The condition system is more flexible than exception systems because instead of providing a two-part division between the code that signals an error[1] and the code that handles it,[2] the condition system splits the responsibilities into three parts--*signaling* a condition, *handling* it, and *restarting*. In this chapter, I'll describe how you could use conditions in part of a hypothetical application for analyzing log files. You'll see how you could use the condition system to allow a low-level function to detect a problem while parsing a log file and signal an error, to allow mid-level code to

provide several possible ways of recovering from such an error, and to allow code at the highest level of the application to define a policy for choosing which recovery strategy to use.

To start, I'll introduce some terminology: *errors*, as I'll use the term, are the consequences of Murphy's law. If something can go wrong, it will: a file that your program needs to read will be missing, a disk that you need to write to will be full, the server you're talking to will crash, or the network will go down. If any of these things happen, it may stop a piece of code from doing what you want. But there's no bug; there's no place in the code that you can fix to make the nonexistent file exist or the disk not be full. However, if the rest of the program is depending on the actions that were going to be taken, then you'd better deal with the error somehow or you *will* have introduced a bug. So, errors aren't caused by bugs, but neglecting to handle an error is almost certainly a bug.

So, what does it mean to handle an error? In a well-written program, each function is a black box hiding its inner workings. Programs are then built out of layers of functions: high-level functions are built on top of the lower-level functions, and so on. This hierarchy of functionality manifests itself at runtime in the form of the call stack: if `high` calls `medium`, which calls `low`, when the flow of control is in `low`, it's also still in `medium` and `high`, that is, they're still on the call stack.

Because each function is a black box, function boundaries are an excellent place to deal with errors. Each function--`low`, for example--has a job to do. Its direct caller--`medium` in this case--is counting on it to do its job. However, an error that prevents it from doing its job puts

all its callers at risk: `medium` called `low` because it needs the work done that `low` does; if that work doesn't get done, `medium` is in trouble. But this means that `medium`'s caller, `high`, is also in trouble--and so on up the call stack to the very top of the program. On the other hand, because each function is a black box, if any of the functions in the call stack can somehow do their job despite underlying errors, then none of the functions above it needs to know there was a problem--all those functions care about is that the function they called somehow did the work expected of it.

In most languages, errors are handled by returning from a failing function and giving the caller the choice of either recovering or failing itself. Some languages use the normal function return mechanism, while languages with exceptions return control by *throwing* or *raising* an exception. Exceptions are a vast improvement over using normal function returns, but both schemes suffer from a common flaw: while searching for a function that can recover, the stack unwinds, which means code that might recover has to do so without the context of what the lower-level code was trying to do when the error actually occurred.

Consider the hypothetical call chain of `high`, `medium`, `low`. If `low` fails and `medium` can't recover, the ball is in `high`'s court. For `high` to handle the error, it must either do its job without any help from `medium` or somehow change things so calling `medium` will work and call it again. The first option is theoretically clean but implies a lot of extra code--a whole extra implementation of whatever it was `medium` was supposed to do. And the further the stack unwinds, the more work that needs to be redone. The second option--patching things up and

retrying--is tricky; for `high` to be able to change the state of the world so a second call into `medium` won't end up causing an error in `low`, it'd need an unseemly knowledge of the inner workings of both `medium` and `low`, contrary to the notion that each function is a black box.

The Lisp Way

Common Lisp's error handling system gives you a way out of this conundrum by letting you separate the code that actually recovers from an error from the code that decides how to recover. Thus, you can put recovery code in low-level functions without committing to actually using any particular recovery strategy, leaving that decision to code in high-level functions.

To get a sense of how this works, let's suppose you're writing an application that reads some sort of textual log file, such as a Web server's log. Somewhere in your application you'll have a function to parse the individual log entries. Let's assume you'll write a function, `parse-log-entry`, that will be passed a string containing the text of a single log entry and that is supposed to return a `log-entry` object representing the entry. This function will be called from a function, `parse-log-file`, that reads a complete log file and returns a list of objects representing all the entries in the file.

To keep things simple, the `parse-log-entry` function will not be required to parse incorrectly formatted entries. It will, however, be able to detect when its input is malformed. But what should it do when it detects bad input? In C you'd return a special value to indicate there was a problem. In Java or Python you'd throw or raise an exception. In Common Lisp, you signal a condition.

Conditions

A *condition* is an object whose class indicates the general nature of the condition and whose instance data carries information about the details of the particular circumstances that lead to the condition being signaled.[3] In this hypothetical log analysis program, you might define a condition class, `malformed-log-entry-error`, that `parse-log-entry` will signal if it's given data it can't parse.

Condition classes are defined with the **DEFINE-CONDITION** macro, which works essentially the same as **DEFCLASS** except that the default superclass of classes defined with **DEFINE-CONDITION** is **CONDITION** rather than **STANDARD-OBJECT**. Slots are specified in the same way, and condition classes can singly and multiply inherit from other classes that descend from **CONDITION**. But for historical reasons, condition classes aren't required to be instances of **STANDARD-OBJECT**, so some of the functions you use with **DEFCLASS**ed classes aren't required to work with conditions. In particular, a condition's slots can't be accessed using **SLOT-VALUE**; you must specify either a `:reader` option or an `:accessor` option for any slot whose value you intend to use. Likewise, new condition objects are created with **MAKE-CONDITION** rather than **MAKE-INSTANCE**. **MAKE-CONDITION** initializes the slots of the new condition based on the `:initarg`s it's passed, but there's no way to further customize a condition's initialization, equivalent to **INITIALIZE-INSTANCE**.[4]

When using the condition system for error handling, you should define your conditions as subclasses of **ERROR**, a subclass of **CONDITION**. Thus, you might define `malformed-log-entry-error`, with a

slot to hold the argument that was passed to `parse-log-entry`, like this:

```
(define-condition malformed-log-entry-error (error)
  ((text :initarg :text :reader text)))
```

Condition Handlers

In `parse-log-entry` you'll signal a `malformed-log-entry-error` if you can't parse the log entry. You signal errors with the function **ERROR**, which calls the lower-level function **SIGNAL** and drops into the debugger if the condition isn't handled. You can call **ERROR** two ways: you can pass it an already instantiated condition object, or you can pass it the name of the condition class and any initargs needed to construct a new condition, and it will instantiate the condition for you. The former is occasionally useful for resignaling an existing condition object, but the latter is more concise. Thus, you could write `parse-log-entry` like this, eliding the details of actually parsing a log entry:

```
(defun parse-log-entry (text)
  (if (well-formed-log-entry-p text)
    (make-instance 'log-entry ...)
    (error 'malformed-log-entry-error :text text)))
```

What happens when the error is signaled depends on the code above `parse-log-entry` on the call stack. To avoid landing in the debugger, you must establish a *condition handler* in one of the functions leading to the call to `parse-log-entry`. When a condition is signaled, the signaling machinery looks through a list of active condition handlers, looking for a handler that can handle the condition being signaled based on the condition's class. Each condition handler consists of a type specifier indicating what types of conditions it can handle and a function that takes a single argument, the condition. At any given moment there can be many active condition handlers established at various levels of the call stack. When a condition is signaled, the signaling machinery finds the most

recently established handler whose type specifier is compatible with the condition being signaled and calls its function, passing it the condition object.

The handler function can then choose whether to handle the condition. The function can decline to handle the condition by simply returning normally, in which case control returns to the **SIGNAL** function, which will search for the next most recently established handler with a compatible type specifier. To handle the condition, the function must transfer control out of **SIGNAL** via a *nonlocal exit*. In the next section, you'll see how a handler can choose where to transfer control. However, many condition handlers simply want to unwind the stack to the place where they were established and then run some code. The macro **HANDLER-CASE** establishes this kind of condition handler. The basic form of a **HANDLER-CASE** is as follows:

```
(handler-case expression
  error-clause*)
```

where each *error-clause* is of the following form:

```
(condition-type ([var]) code)
```

If the *expression* returns normally, then its value is returned by the **HANDLER-CASE**. The body of a **HANDLER-CASE** must be a single expression; you can use **PROGN** to combine several expressions into a single form. If, however, the expression signals a condition that's an instance of any of the *condition-type*s specified in any *error-clause*, then the code in the appropriate error clause is executed and its value returned by the **HANDLER-CASE**. The *var*, if included, is the name of the variable that will hold the condition object when the handler

code is executed. If the code doesn't need to access the condition object, you can omit the variable name.

For instance, one way to handle the `malformed-log-entry-error` signaled by `parse-log-entry` in its caller, `parse-log-file`, would be to skip the malformed entry. In the following function, the **HANDLER-CASE** expression will either return the value returned by `parse-log-entry` or return **NIL** if a `malformed-log-entry-error` is signaled. (The `it` in the **LOOP** clause `collect it` is another **LOOP** keyword, which refers to the value of the most recently evaluated conditional test, in this case the value of `entry`.)

```
(defun parse-log-file (file)
  (with-open-file (in file :direction :input)
    (loop for text = (read-line in nil nil) while text
       for entry = (handler-case (parse-log-entry text)
                     (malformed-log-entry-error () nil))
       when entry collect it)))
```

When `parse-log-entry` returns normally, its value will be assigned to `entry` and collected by the **LOOP**. But if `parse-log-entry` signals a `malformed-log-entry-error`, then the error clause will return **NIL**, which won't be collected.

JAVA-STYLE EXCEPTON HANDLING

HANDLER-CASE is the nearest analog in Common Lisp to Java- or Python-style exception handling. Where you might write this in Java:

```
try {
  doStuff();
  doMoreStuff();
} catch (SomeException se) {
  recover(se);
}
```

or this in Python:

```
    try:
      doStuff()
      doMoreStuff()
    except SomeException, se:
      recover(se)
```

in Common Lisp you'd write this:

```
    (handler-case
        (progn
          (do-stuff)
          (do-more-stuff))
      (some-exception (se) (recover se)))
```

This version of `parse-log-file` has one serious deficiency: it's doing too much. As its name suggests, the job of `parse-log-file` is to parse the file and produce a list of `log-entry` objects; if it can't, it's not its place to decide what to do instead. What if you want to use `parse-log-file` in an application that wants to tell the user that the log file is corrupted or one that wants to recover from malformed entries by fixing them up and re-parsing them? Or maybe an application is fine with skipping them but only until a certain number of corrupted entries have been seen.

You could try to fix this problem by moving the **`HANDLER-CASE`** to a higher-level function. However, then you'd have no way to implement the current policy of skipping individual entries--when the error was signaled, the stack would be unwound all the way to the higher-level function, abandoning the parsing of the log file altogether. What you want is a way to provide the current recovery strategy without requiring that it always be used.

Restarts

The condition system lets you do this by splitting the error handling code into two parts. You place code that actually recovers from errors into *restarts*, and condition handlers can then handle a condition by invoking an appropriate restart. You can place restart code in mid- or low-level functions, such as `parse-log-file` or `parse-log-entry`, while moving the condition handlers into the upper levels of the application.

To change `parse-log-file` so it establishes a restart instead of a condition handler, you can change the **`HANDLER-CASE`** to a **`RESTART-CASE`**. The form of **`RESTART-CASE`** is quite similar to a **`HANDLER-CASE`** except the names of restarts are just names, not necessarily the names of condition types. In general, a restart name should describe the action the restart takes. In `parse-log-file`, you can call the restart `skip-log-entry` since that's what it does. The new version will look like this:

```
(defun parse-log-file (file)
  (with-open-file (in file :direction :input)
    (loop for text = (read-line in nil nil) while text
       for entry = (restart-case (parse-log-entry text)
                     (skip-log-entry () nil))
       when entry collect it)))
```

If you invoke this version of `parse-log-file` on a log file containing corrupted entries, it won't handle the error directly; you'll end up in the debugger. However, there among the various restarts presented by the debugger will be one called `skip-log-entry`, which, if you choose it, will cause `parse-log-file` to continue on its way as before.

To avoid ending up in the debugger, you can establish a condition handler that invokes the `skip-log-entry` restart automatically.

The advantage of establishing a restart rather than having `parse-log-file` handle the error directly is it makes `parse-log-file` usable in more situations. The higher-level code that invokes `parse-log-file` doesn't have to invoke the `skip-log-entry` restart. It can choose to handle the error at a higher level. Or, as I'll show in the next section, you can add restarts to `parse-log-entry` to provide other recovery strategies, and then condition handlers can choose which strategy they want to use.

But before I can talk about that, you need to see how to set up a condition handler that will invoke the `skip-log-entry` restart. You can set up the handler anywhere in the chain of calls leading to `parse-log-file`. This may be quite high up in your application, not necessarily in `parse-log-file`'s direct caller. For instance, suppose the main entry point to your application is a function, `log-analyzer`, that finds a bunch of logs and analyzes them with the function `analyze-log`, which eventually leads to a call to `parse-log-file`. Without any error handling, it might look like this:

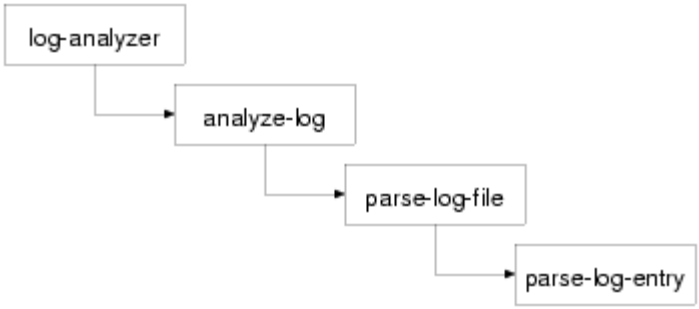
```
(defun log-analyzer ()
  (dolist (log (find-all-logs))
    (analyze-log log)))
```

The job of `analyze-log` is to call, directly or indirectly, `parse-log-file` and then do something with the list of log entries returned. An extremely simple version might look like this:

```
(defun analyze-log (log)
  (dolist (entry (parse-log-file log))
    (analyze-entry entry)))
```

where the function `analyze-entry` is presumably responsible for extracting whatever information you care about from each log entry and stashing it away somewhere.

Thus, the path from the top-level function, `log-analyzer`, to `parse-log-entry`, which actually signals an error, is as follows:



Assuming you always want to skip malformed log entries, you could change this function to establish a condition handler that invokes the `skip-log-entry` restart for you. However, you can't use **`HANDLER-CASE`** to establish the condition handler because then the stack would be unwound to the function where the **`HANDLER-CASE`** appears. Instead, you need to use the lower-level macro **`HANDLER-BIND`**. The basic form of **`HANDLER-BIND`** is as follows:

```
(handler-bind (binding*) form*)
```

where each binding is a list of a condition type and a handler function of one argument. After the handler bindings, the body of the **`HANDLER-BIND`** can contain any number of forms. Unlike the handler code in **`HANDLER-CASE`**, the handler code must be a function object, and it must accept a single argument. A more important difference between **`HANDLER-BIND`** and **`HANDLER-CASE`** is that the handler function bound by **`HANDLER-BIND`** will be run without unwinding the stack--the

flow of control will still be in the call to `parse-log-entry` when this function is called. The call to **`INVOKE-RESTART`** will find and invoke the most recently bound restart with the given name. So you can add a handler to `log-analyzer` that will invoke the `skip-log-entry` restart established in `parse-log-file` like this:[5]

```
(defun log-analyzer ()
  (handler-bind ((malformed-log-entry-error
                  #'(lambda (c)
                      (invoke-restart 'skip-log-entry))))
    (dolist (log (find-all-logs))
      (analyze-log log))))
```

In this **`HANDLER-BIND`**, the handler function is an anonymous function that invokes the restart `skip-log-entry`. You could also define a named function that does the same thing and bind it instead. In fact, a common practice when defining a restart is to define a function, with the same name and taking a single argument, the condition, that invokes the eponymous restart. Such functions are called *restart functions*. You could define a restart function for `skip-log-entry` like this:

```
(defun skip-log-entry (c)
  (invoke-restart 'skip-log-entry))
```

Then you could change the definition of `log-analyzer` to this:

```
(defun log-analyzer ()
  (handler-bind ((malformed-log-entry-error #'skip-log-entry))
    (dolist (log (find-all-logs))
      (analyze-log log))))
```

As written, the `skip-log-entry` restart function assumes that a `skip-log-entry` restart has been established. If a `malformed-log-entry-error` is ever signaled by code called from `log-analyzer` without a `skip-log-entry` having been established, the call to **`INVOKE-RESTART`** will signal a **`CONTROL-ERROR`** when it fails to find

the `skip-log-entry` restart. If you want to allow for the possibility that a `malformed-log-entry-error` might be signaled from code that doesn't have a `skip-log-entry` restart established, you could change the `skip-log-entry` function to this:

```
(defun skip-log-entry (c)
  (let ((restart (find-restart 'skip-log-entry)))
    (when restart (invoke-restart restart))))
```

FIND-RESTART looks for a restart with a given name and returns an object representing the restart if the restart is found and **NIL** if not. You can invoke the restart by passing the restart object to **INVOKE-RESTART**. Thus, when `skip-log-entry` is bound with **HANDLER-BIND**, it will handle the condition by invoking the `skip-log-entry` restart if one is available and otherwise will return normally, giving other condition handlers, bound higher on the stack, a chance to handle the condition.

Providing Multiple Restarts

Since restarts must be explicitly invoked to have any effect, you can define multiple restarts, each providing a different recovery strategy. As I mentioned earlier, not all log-parsing applications will necessarily want to skip malformed entries. Some applications might want `parse-log-file` to include a special kind of object representing malformed entries in the list of `log-entry` objects; other applications may have some way to repair a malformed entry and may want a way to pass the fixed entry back to `parse-log-entry`.

To allow more complex recovery protocols, restarts can take arbitrary arguments, which are passed in the call to **`INVOKE-RESTART`**. You can provide support for both the recovery strategies I just mentioned by adding two restarts to `parse-log-entry`, each of which takes a single argument. One simply returns the value it's passed as the return value of `parse-log-entry`, while the other tries to parse its argument in the place of the original log entry.

```
(defun parse-log-entry (text)
  (if (well-formed-log-entry-p text)
    (make-instance 'log-entry ...)
    (restart-case (error 'malformed-log-entry-error :text text)
      (use-value (value) value)
      (reparse-entry (fixed-text) (parse-log-entry fixed-text)))))
```

The name **`USE-VALUE`** is a standard name for this kind of restart. Common Lisp defines a restart function for **`USE-VALUE`** similar to the `skip-log-entry` function you just defined. So, if you wanted to change the policy on malformed entries to one that created an instance of `malformed-log-entry`, you could change `log-analyzer` to

this (assuming the existence of a `malformed-log-entry` class with a `:text` initarg):

```
(defun log-analyzer ()
  (handler-bind ((malformed-log-entry-error
                  #'(lambda (c)
                      (use-value
                       (make-instance 'malformed-log-entry :text (text c))))))
    (dolist (log (find-all-logs))
      (analyze-log log))))
```

You could also have put these new restarts into `parse-log-file` instead of `parse-log-entry`. However, you generally want to put restarts in the lowest-level code possible. It wouldn't, though, be appropriate to move the `skip-log-entry` restart into `parse-log-entry` since that would cause `parse-log-entry` to sometimes return normally with **`NIL`**, the very thing you started out trying to avoid. And it'd be an equally bad idea to remove the `skip-log-entry` restart on the theory that the condition handler could get the same effect by invoking the `use-value` restart with **`NIL`** as the argument; that would require the condition handler to have intimate knowledge of how the `parse-log-file` works. As it stands, the `skip-log-entry` is a properly abstracted part of the log-parsing API.

Other Uses for Conditions

While conditions are mainly used for error handling, they can be used for other purposes--you can use conditions, condition handlers, and restarts to build a variety of protocols between low- and high-level code. The key to understanding the potential of conditions is to understand that merely signaling a condition has no effect on the flow of control.

The primitive signaling function `SIGNAL` implements the mechanism of searching for an applicable condition handler and invoking its handler function. The reason a handler can decline to handle a condition by returning normally is because the call to the handler function is just a regular function call--when the handler returns, control passes back to `SIGNAL`, which then looks for another, less recently bound handler that can handle the condition. If `SIGNAL` runs out of condition handlers before the condition is handled, it also returns normally.

The `ERROR` function you've been using calls `SIGNAL`. If the error is handled by a condition handler that transfers control via `HANDLER-CASE` or by invoking a restart, then the call to `SIGNAL` never returns. But if `SIGNAL` returns, `ERROR` invokes the debugger by calling the function stored in `*DEBUGGER-HOOK*`. Thus, a call to `ERROR` can never return normally; the condition must be handled either by a condition handler or in the debugger.

Another condition signaling function, `WARN`, provides an example of a different kind of protocol built on the condition system. Like `ERROR`,

WARN calls **SIGNAL** to signal a condition. But if **SIGNAL** returns, **WARN** doesn't invoke the debugger--it prints the condition to ***ERROR-OUTPUT*** and returns **NIL**, allowing its caller to proceed. **WARN** also establishes a restart, **MUFFLE-WARNING**, around the call to **SIGNAL** that can be used by a condition handler to make **WARN** return without printing anything. The restart function **MUFFLE-WARNING** finds and invokes its eponymous restart, signaling a **CONTROL-ERROR** if no such restart is available. Of course, a condition signaled with **WARN** could also be handled in some other way--a condition handler could "promote" a warning to an error by handling it as if it were an error.

For instance, in the log-parsing application, if there were ways a log entry could be slightly malformed but still parsable, you could write `parse-log-entry` to go ahead and parse the slightly defective entries but to signal a condition with **WARN** when it did. Then the larger application could choose to let the warning print, to muffle the warning, or to treat the warning like an error, recovering the same way it would from a `malformed-log-entry-error`.

A third error-signaling function, **CERROR**, provides yet another protocol. Like **ERROR**, **CERROR** will drop you into the debugger if the condition it signals isn't handled. But like **WARN**, it establishes a restart before it signals the condition. The restart, **CONTINUE**, causes **CERROR** to return normally--if the restart is invoked by a condition handler, it will keep you out of the debugger altogether. Otherwise, you can use the restart once you're in the debugger to resume the computation immediately after the call to **CERROR**. The function **CONTINUE** finds and invokes the **CONTINUE** restart if it's available and returns **NIL** otherwise.

You can also build your own protocols on `SIGNAL`--whenever low-level code needs to communicate information back up the call stack to higher-level code, the condition mechanism is a reasonable mechanism to use. But for most purposes, one of the standard error or warning protocols should suffice.

You'll use the condition system in future practical chapters, both for regular error handling and, in Chapter 25, to help in handling a tricky corner case of parsing ID3 files. Unfortunately, it's the fate of error handling to always get short shrift in programming texts--proper error handling, or lack thereof, is often the biggest difference between illustrative code and hardened, production-quality code. The trick to writing the latter has more to do with adopting a particularly rigorous way of thinking about software than with the details of any particular programming language constructs. That said, if your goal is to write that kind of software, you'll find the Common Lisp condition system is an excellent tool for writing robust code and one that fits quite nicely into Common Lisp's incremental development style.

Writing Robust Software

For information on writing robust software, you could do worse than to start by finding a copy of *Software Reliability* (John Wiley & Sons, 1976) by Glenford J. Meyers. Bertrand Meyer's writings on Design By Contract also provide a useful way of thinking about software correctness. For instance, see Chapters 11 and 12 of his *Object-Oriented Software Construction* (Prentice Hall, 1997). Keep in mind, however, that Bertrand Meyer is the inventor of Eiffel, a statically typed bondage and discipline language in the Algol/Ada school. While he has a lot of smart things to say about object orientation and software reliability, there's a fairly wide gap between his view of programming and The Lisp Way. Finally, for an

excellent overview of the larger issues surrounding building fault-tolerant systems, see Chapter 3 of the classic *Transaction Processing: Concepts and Techniques* (Morgan Kaufmann, 1993) by Jim Gray and Andreas Reuter.

In the next chapter I'll give a quick overview of some of the 25 special operators you haven't had a chance to use yet, at least not directly.

---

[1] *Throws* or *raises* an exception in Java/Python terms

[2] *Catches* the exception in Java/Python terms

[3] In this respect, a condition is a lot like an exception in Java or Python except not all conditions represent an error or *exceptional* situation.

[4] In some Common Lisp implementations, conditions are defined as subclasses of **STANDARD-OBJECT**, in which case **SLOT-VALUE**, **MAKE-INSTANCE**, and **INITIALIZE-INSTANCE** will work, but it's not portable to rely on it.

[5] The compiler may complain if the parameter is never used. You can silence that warning by adding a declaration `(declare (ignore c))` as the first expression in the **LAMBDA** body.

Copyright © 2003-2005, Peter Seibel

20. The Special Operators

In a way, the most impressive aspect of the condition system covered in the previous chapter is that if it wasn't already part of the language, it could be written entirely as a user-level library. This is possible because Common Lisp's special operators--while none touches directly on signaling or handling conditions--provide enough access to the underlying machinery of the language to be able to do things such as control the unwinding of the stack.

In previous chapters I've discussed the most frequently used special operators, but it's worth being familiar with the others for two reasons. First, some of the infrequently used special operators are used infrequently simply because whatever need they address doesn't arise that often. It's good to be familiar with these special operators so when one of them is called for, you'll at least know it exists. Second, because the 25 special operators--along with the basic rule for evaluating function calls and the built-in data types--provide the foundation for the rest of the language, a passing familiarity with them will help you understand how the language works.

In this chapter, I'll discuss all the special operators, some briefly and some at length, so you can see how they fit together. I'll point out which ones you can expect to use directly in your own code, which ones serve as the basis for other constructs that you use all the time, and which ones you'll rarely use directly but which can be handy in macro-generated code.

Controlling Evaluation

The first category of special operators contains the three operators that provide basic control over the evaluation of forms. They're QUOTE, IF, and PROGN, and I've discussed them all already. However, it's worth noting how each of these special operators provides one fundamental kind of control over the evaluation of one or more forms. QUOTE prevents evaluation altogether and allows you to get at s-expressions as data. IF provides the fundamental boolean choice operation from which all other conditional execution constructs can be built.[1] And PROGN provides the ability to sequence a number of forms.

Manipulating the Lexical Environment

The largest class of special operators contains the operators that manipulate and access the *lexical environment*. LET and LET*, which I've already discussed, are examples of special operators that manipulate the lexical environment since they can introduce new lexical bindings for variables. Any construct, such as a DO or DOTIMES, that binds lexical variables will have to expand into a LET or LET*.[2] The SETQ special operator is one that accesses the lexical environment since it can be used to set variables whose bindings were created by LET and LET*.

Variables, however, aren't the only thing that can be named within a lexical scope. While most functions are defined globally with DEFUN, it's also possible to create local functions with the special operators FLET and LABELS, local macros with MACROLET, and a special kind of macro, called a *symbol macro*, with SYMBOL-MACROLET.

Much like LET allows you to introduce a lexical variable whose scope is the body of the LET, FLET and LABELS let you define a function that can be referred to only within the scope of the FLET or LABELS form. These special operators are handy when you need a local function that's a bit too complex to define inline as a LAMBDA expression or that you need to use more than once. Both have the same basic form, which looks like this:

```
(flet (function-definition*)
  body-form*)
```

and like this:

```
(labels (function-definition*)
  body-form*)
```

where each *function-definition* has the following form:

```
(name (parameter*) form*)
```

The difference between **FLET** and **LABELS** is that the names of the functions defined with **FLET** can be used only in the body of the **FLET**, while the names introduced by **LABELS** can be used immediately, including in the bodies of the functions defined by the **LABELS**. Thus, **LABELS** can define recursive functions, while **FLET** can't. It might seem limiting that **FLET** can't be used to define recursive functions, but Common Lisp provides both **FLET** and **LABELS** because sometimes it's useful to be able to write local functions that can call another function of the same name, either a globally defined function or a local function from an enclosing scope.

Within the body of a **FLET** or **LABELS**, you can use the names of the functions defined just like any other function, including with the **FUNCTION** special operator. Since you can use **FUNCTION** to get the function object representing a function defined with **FLET** or **LABELS**, and since a **FLET** or **LABELS** can be in the scope of other binding forms such as **LET**s, these functions can be closures.

Because the local functions can refer to variables from the enclosing scope, they can often be written to take fewer parameters than the equivalent helper functions. This is particularly handy when you need to pass a function that takes a single argument as a functional parameter. For example, in the following function, which you'll see again in Chapter 25, the **FLET**ed function, `count-version`, takes a

single argument, as required by `walk-directory`, but can also use the variable `versions`, introduced by the enclosing **`LET`**:

```
(defun count-versions (dir)
  (let ((versions (mapcar #'(lambda (x) (cons x 0)) '(2 3 4))))
    (flet ((count-version (file)
             (incf (cdr (assoc (major-version (read-id3 file)) versions)))))
      (walk-directory dir #'count-version :test #'mp3-p))
    versions))
```

This function could also be written using an anonymous function in the place of the **`FLET`**ed `count-version`, but giving the function a meaningful name makes it a bit easier to read.

And when a helper function needs to recurse, an anonymous function just won't do.[3] When you don't want to define a recursive helper function as a global function, you can use **`LABELS`**. For example, the following function, `collect-leaves`, uses the recursive helper function `walk` to walk a tree and gather all the atoms in the tree into a list, which `collect-leaves` then returns (after reversing it):

```
(defun collect-leaves (tree)
  (let ((leaves ()))
    (labels ((walk (tree)
               (cond
                 ((null tree))
                 ((atom tree) (push tree leaves))
                 (t (walk (car tree))
                    (walk (cdr tree))))))
      (walk tree))
    (nreverse leaves)))
```

Notice again how, within the `walk` function, you can refer to the variable, `leaves`, introduced by the enclosing **`LET`**.

`FLET` and **`LABELS`** are also useful operations to use in macro expansions--a macro can expand into code that contains a **`FLET`** or **`LABELS`** to create functions that can be used within the body of the macro. This technique can be used either to introduce functions that

the user of the macro will call or simply as a way of organizing the code generated by the macro. This, for instance, is how a function such as CALL-NEXT-METHOD, which can be used only within a method definition, might be defined.

A near relative to FLET and LABELS is the special operator MACROLET, which you can use to define local macros. Local macros work just like global macros defined with DEFMACRO except without cluttering the global namespace. When a MACROLET form is evaluated, the body forms are evaluated with the local macro definitions in effect and possibly shadowing global function and macro definitions or local definitions from enclosing forms. Like FLET and LABELS, MACROLET can be used directly, but it's also a handy target for macro-generated code--by wrapping some user-supplied code in a MACROLET, a macro can provide constructs that can be used only within that code or can shadow a globally defined macro. You'll see an example of this latter use of MACROLET in Chapter 31.

Finally, one last macro-defining special operator is SYMBOL-MACROLET, which defines a special kind of macro called, appropriately enough, a *symbol macro*. Symbol macros are like regular macros except they can't take arguments and are referred to with a plain symbol rather than a list form. In other words, after you've defined a symbol macro with a particular name, any use of that symbol in a value position will be expanded and the resulting form evaluated in its place. This is how macros such as WITH-SLOTS and WITH-ACCESSORS are able to define "variables" that access the state of a particular object under the covers. For instance, the following WITH-SLOTS form:

```
(with-slots (x y z) foo (list x y z)))
```

might expand into this code that uses **SYMBOL-MACROLET**:

```
(let ((#:g149 foo))
  (symbol-macrolet
      ((x (slot-value #:g149 'x))
       (y (slot-value #:g149 'y))
       (z (slot-value #:g149 'z)))
    (list x y z)))
```

When the expression `(list x y z)` is evaluated, the symbols `x`, `y`, and `z` will be replaced with their expansions, such as `(slot-value #:g149 'x)`.[4]

Symbol macros are most often local, defined with **SYMBOL-MACROLET**, but Common Lisp also provides a macro **DEFINE-SYMBOL-MACRO** that defines a global symbol macro. A symbol macro defined with **SYMBOL-MACROLET** shadows other symbol macros of the same name defined with **DEFINE-SYMBOL-MACRO** or enclosing **SYMBOL-MACROLET** forms.

Local Flow of Control

The next four special operators I'll discuss also create and use names in the lexical environment but for the purposes of altering the flow of control rather than defining new functions and macros. I've mentioned all four of these special operators in passing because they provide the underlying mechanisms used by other language features. They're BLOCK, RETURN-FROM, TAGBODY, and GO. The first two, BLOCK and RETURN-FROM, are used together to write code that returns immediately from a section of code--I discussed RETURN-FROM in Chapter 5 as a way to return immediately from a function, but it's more general than that. The other two, TAGBODY and GO, provide a quite low-level goto construct that's the basis for all the higher-level looping constructs you've already seen.

The basic skeleton of a BLOCK form is this:

```
(block name
  form*)
```

The *name* is a symbol, and the *forms* are Lisp forms. The forms are evaluated in order, and the value of the last form is returned as the value of the BLOCK unless a RETURN-FROM is used to return from the block early. A RETURN-FROM form, as you saw in Chapter 5, consists of the name of the block to return from and, optionally, a form that provides a value to return. When a RETURN-FROM is evaluated, it causes the named BLOCK to return immediately. If RETURN-FROM is called with a return value form, the BLOCK will return the resulting value; otherwise, the BLOCK evaluates to NIL.

A BLOCK name can be any symbol, which includes NIL. Many of the standard control construct macros, such as DO, DOTIMES, and DOLIST, generate an expansion consisting of a BLOCK named NIL. This allows you to use the RETURN macro, which is a bit of syntactic sugar for `(return-from nil ...)`, to break out of such loops. Thus, the following loop will print at most ten random numbers, stopping as soon as it gets a number greater than 50:

```
(dotimes (i 10)
  (let ((answer (random 100)))
    (print answer)
    (if (> answer 50) (return))))
```

Function-defining macros such as DEFUN, FLET, and LABELS, on the other hand, wrap their bodies in a BLOCK with the same name as the function. That's why you can use RETURN-FROM to return from a function.

TAGBODY and GO have a similar relationship to each other as BLOCK and RETURN-FROM: a TAGBODY form defines a context in which names are defined that can be used by GO. The skeleton of a TAGBODY is as follows:

```
(tagbody
  tag-or-compound-form*)
```

where each *tag-or-compound-form* is either a symbol, called a *tag*, or a nonempty list form. The list forms are evaluated in order and the tags ignored, except as I'll discuss in a moment. After the last form of the TAGBODY is evaluated, the TAGBODY returns NIL. Anywhere within the lexical scope of the TAGBODY you can use the GO special operator to jump immediately to any of the tags, and evaluation will resume

with the form following the tag. For instance, you can write a trivial infinite loop with **TAGBODY** and **GO** like this:

```
(tagbody
 top
   (print 'hello)
   (go top))
```

Note that while the tag names must appear at the top level of the **TAGBODY**, not nested within other forms, the **GO** special operator can appear anywhere within the scope of the **TAGBODY**. This means you could write a loop that loops a random number of times like this:

```
(tagbody
 top
   (print 'hello)
   (when (plusp (random 10)) (go top)))
```

An even sillier example of **TAGBODY**, which shows you can have multiple tags in a single **TAGBODY**, looks like this:

```
(tagbody
 a (print 'a) (if (zerop (random 2)) (go c))
 b (print 'b) (if (zerop (random 2)) (go a))
 c (print 'c) (if (zerop (random 2)) (go b)))
```

This form will jump around randomly printing *a*s, *b*s, and *c*s until eventually the last **RANDOM** expression returns 1 and the control falls off the end of the **TAGBODY**.

TAGBODY is rarely used directly since it's almost always easier to write iterative constructs in terms of the existing looping macros. It's handy, however, for translating algorithms written in other languages into Common Lisp, either automatically or manually. An example of an automatic translation tool is the FORTRAN-to-Common Lisp translator, f2cl, that translates FORTRAN source code into Common Lisp in order to make various FORTRAN libraries available to

Common Lisp programmers. Since many FORTRAN libraries were written before the structured programming revolution, they're full of gotos. The f2cl compiler can simply translate those gotos to `GO`s within appropriate `TAGBODY`s.[5]

Similarly, `TAGBODY` and `GO` can be handy when translating algorithms described in prose or by flowcharts--for instance, in Donald Knuth's classic series *The Art of Computer Programming*, he describes algorithms using a "recipe" format: step 1, do this; step 2, do that; step 3, go back to step 2; and so on. For example, on page 142 of *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Third Edition (Addison-Wesley, 1998), he describes Algorithm S, which you'll use in Chapter 27, in this form:

> Algorithm S (Selection sampling technique). To select n records at random from a set of N, where 0 < n <= N.
>
> S1. [Initialize.] Set t <-- 0, m <-- 0. (During this algorithm, m represents the number of records selected so far, and t is the total number of input records that we have dealt with.)
>
> S2. [Generate U.] Generate a random number U, uniformly distributed between zero and one.
>
> S3. [Test.] If (N - t)U >= n - m, go to step S5.
>
> S4. [Select.] Select the next record for the sample, and increase m and t by 1. If m < n, go to step S2; otherwise the sample is complete and the algorithm terminates.
>
> S5. [Skip.] Skip the next record (do not include it in the sample), increase t by 1, and go back to step S2.

This description can be easily translated into a Common Lisp function, after renaming a few variables, as follows:

```
(defun algorithm-s (n max) ; max is N in Knuth's algorithm
  (let (seen               ; t in Knuth's algorithm
        selected           ; m in Knuth's algorithm
        u                  ; U in Knuth's algorithm
        (records ()))      ; the list where we save the records selected
    (tagbody
     s1
       (setf seen 0)
       (setf selected 0)
     s2
       (setf u (random 1.0))
     s3
       (when (>= (* (- max seen) u) (- n selected)) (go s5))
     s4
       (push seen records)
       (incf selected)
       (incf seen)
       (if (< selected n)
           (go s2)
           (return-from algorithm-s (nreverse records)))
     s5
       (incf seen)
       (go s2))))
```

It's not the prettiest code, but it's easy to verify that it's a faithful translation of Knuth's algorithm. But, this code, unlike Knuth's prose description, can be run and tested. Then you can start refactoring, checking after each change that the function still works.[6]

After pushing the pieces around a bit, you might end up with something like this:

```
(defun algorithm-s (n max)
  (loop for seen from 0
     when (< (* (- max seen) (random 1.0)) n)
     collect seen and do (decf n)
     until (zerop n)))
```

While it may not be immediately obvious that this code correctly implements Algorithm S, if you got here via a series of functions that all behave identically to the original literal translation of Knuth's recipe, you'd have good reason to believe it's correct.

Unwinding the Stack

Another aspect of the language that special operators give you control over is the behavior of the call stack. For instance, while you normally use **BLOCK** and **TAGBODY** to manage the flow of control within a single function, you can also use them, in conjunction with closures, to force an immediate nonlocal return from a function further down on the stack. That's because **BLOCK** names and **TAGBODY** tags can be closed over by any code within the lexical scope of the **BLOCK** or **TAGBODY**. For example, consider this function:

```
(defun foo ()
  (format t "Entering foo~%")
  (block a
    (format t " Entering BLOCK~%")
    (bar #'(lambda () (return-from a)))
    (format t " Leaving BLOCK~%"))
  (format t "Leaving foo~%"))
```

The anonymous function passed to `bar` uses **RETURN-FROM** to return from the **BLOCK**. But that **RETURN-FROM** doesn't get evaluated until the anonymous function is invoked with **FUNCALL** or **APPLY**. Now suppose `bar` looks like this:

```
(defun bar (fn)
  (format t "  Entering bar~%")
  (baz fn)
  (format t "  Leaving bar~%"))
```

Still, the anonymous function isn't invoked. Now look at `baz`.

```
(defun baz (fn)
  (format t "   Entering baz~%")
  (funcall fn)
  (format t "   Leaving baz~%"))
```

Finally the function is invoked. But what does it mean to **RETURN-FROM** a block that's several layers up on the call stack? Turns out it works

fine--the stack is unwound back to the frame where the **BLOCK** was established and control returns from the **BLOCK**. The **FORMAT** expressions in `foo`, `bar`, and `baz` show this:

```
CL-USER> (foo)
Entering foo
 Entering BLOCK
  Entering bar
   Entering baz
Leaving foo
NIL
```

Note that the only "Leaving . . ." message that prints is the one that appears after the **BLOCK** in `foo`.

Because the names of blocks are lexically scoped, a **RETURN-FROM** always returns from the smallest enclosing **BLOCK** in the lexical environment where the **RETURN-FROM** form appears even if the **RETURN-FROM** is executed in a different dynamic context. For instance, `bar` could also contain a **BLOCK** named `a`, like this:

```
(defun bar (fn)
  (format t "  Entering bar~%")
  (block a (baz fn))
  (format t "  Leaving bar~%"))
```

This extra **BLOCK** won't change the behavior of `foo` at all--the name `a` is resolved lexically, at compile time, not dynamically, so the intervening block has no effect on the **RETURN-FROM**. Conversely, the name of a **BLOCK** can be used only by **RETURN-FROM**s appearing within the lexical scope of the **BLOCK**; there's no way for code outside the block to return from the block except by invoking a closure that closes over a **RETURN-FROM** from the lexical scope of the **BLOCK**.

TAGBODY and **GO** work the same way, in this regard, as **BLOCK** and **RETURN-FROM**. When you invoke a closure that contains a **GO** form, if

the GO is evaluated, the stack will unwind back to the appropriate TAGBODY and then jump to the specified tag.

BLOCK names and TAGBODY tags, however, differ from lexical variable bindings in one important way. As I discussed in Chapter 6, lexical bindings have indefinite extent, meaning the bindings can stick around even after the binding form has returned. BLOCKs and TAGBODYs, on the other hand, have dynamic extent--you can RETURN-FROM a BLOCK or GO to a TAGBODY tag only while the BLOCK or TAGBODY is on the call stack. In other words, a closure that captures a block name or TAGBODY tag can be passed *down* the stack to be invoked later, but it can't be returned *up* the stack. If you invoke a closure that tries to RETURN-FROM a BLOCK, after the BLOCK itself has returned, you'll get an error. Likewise, trying to GO to a TAGBODY that no longer exists will cause an error.[7]

It's unlikely you'll need to use BLOCK and TAGBODY yourself for this kind of stack unwinding. But you'll likely be using them indirectly whenever you use the condition system, so understanding how they work should help you understand better what exactly, for instance, invoking a restart is doing.[8]

CATCH and THROW are another pair of special operators that can force the stack to unwind. You'll use these operators even less often than the others mentioned so far--they're holdovers from earlier Lisp dialects that didn't have Common Lisp's condition system. They definitely shouldn't be confused with `try`/`catch` and `try`/`except` constructs from languages such as Java and Python.

CATCH and **THROW** are the dynamic counterparts of **BLOCK** and **RETURN-FROM**. That is, you wrap **CATCH** around a body of code and then use **THROW** to cause the **CATCH** form to return immediately with a specified value. The difference is that the association between a **CATCH** and **THROW** is established dynamically--instead of a lexically scoped name, the label for a **CATCH** is an object, called a *catch tag*, and any **THROW** evaluated within the dynamic extent of the **CATCH** that throws that object will unwind the stack back to the **CATCH** form and cause it to return immediately. Thus, you can write a version of the `foo`, `bar`, and `baz` functions from before using **CATCH** and **THROW** instead of **BLOCK** and **RETURN-FROM** like this:

```
(defparameter *obj* (cons nil nil)) ; i.e. some arbitrary object

(defun foo ()
  (format t "Entering foo~%")
  (catch *obj*
    (format t " Entering CATCH~%")
    (bar)
    (format t " Leaving CATCH~%"))
  (format t "Leaving foo~%"))

(defun bar ()
  (format t "  Entering bar~%")
  (baz)
  (format t "  Leaving bar~%"))

(defun baz ()
  (format t "   Entering baz~%")
  (throw *obj* nil)
  (format t "   Leaving baz~%"))
```

Notice how it isn't necessary to pass a closure down the stack--`baz` can call **THROW** directly. The result is quite similar to the earlier version.

```
CL-USER> (foo)
Entering foo
 Entering CATCH
  Entering bar
   Entering baz
Leaving foo
NIL
```

However, CATCH and THROW are almost *too* dynamic. In both the CATCH and the THROW, the tag form is evaluated, which means their values are both determined at runtime. Thus, if some code in `bar` reassigned or rebound `*obj*`, the THROW in `baz` wouldn't throw to the same CATCH. This makes CATCH and THROW much harder to reason about than BLOCK and RETURN-FROM. The only advantage, which the version of `foo`, `bar`, and `baz` that use CATCH and THROW demonstrates, is there's no need to pass down a closure in order for low-level code to return from a CATCH--any code that runs within the dynamic extent of a CATCH can cause it to return by throwing the right object.

In older Lisp dialects that didn't have anything like Common Lisp's condition system, CATCH and THROW were used for error handling. However, to keep them manageable, the catch tags were usually just quoted symbols, so you *could* tell by looking at a CATCH and a THROW whether they would hook up at runtime. In Common Lisp you'll rarely have any call to use CATCH and THROW since the condition system is so much more flexible.

The last special operator related to controlling the stack is another one I've mentioned in passing before--UNWIND-PROTECT. UNWIND-PROTECT lets you control what happens as the stack unwinds--to make sure that certain code always runs regardless of how control leaves the scope of the UNWIND-PROTECT, whether by a normal return, by a restart being invoked, or by any of the ways discussed in this section.[9] The basic skeleton of UNWIND-PROTECT looks like this:

```
(unwind-protect protected-form
  cleanup-form*)
```

The single *protected-form* is evaluated, and then, regardless of how it returns, the *cleanup-forms* are evaluated. If the *protected-form* returns normally, then whatever it returns is returned from the **UNWIND-PROTECT** after the cleanup forms run. The cleanup forms are evaluated in the same dynamic environment as the **UNWIND-PROTECT**, so the same dynamic variable bindings, restarts, and condition handlers will be visible to code in cleanup forms as were visible just before the **UNWIND-PROTECT**.

You'll occasionally use **UNWIND-PROTECT** directly. More often you'll use it as the basis for `WITH-` style macros, similar to **WITH-OPEN-FILE**, that evaluate any number of body forms in a context where they have access to some resource that needs to be cleaned up after they're done, regardless of whether they return normally or bail via a restart or other nonlocal exit. For example, if you were writing a database library that defined functions `open-connection` and `close-connection`, you might write a macro like this:[10]

```
(defmacro with-database-connection ((var &rest open-args) &body body)
  `(let ((,var (open-connection ,@open-args)))
     (unwind-protect (progn ,@body)
       (close-connection ,var))))
```

which lets you write code like this:

```
(with-database-connection (conn :host "foo" :user "scott" :password "tiger")
  (do-stuff conn)
  (do-more-stuff conn))
```

and not have to worry about closing the database connection, since the **UNWIND-PROTECT** will make sure it gets closed no matter what happens in the body of the `with-database-connection` form.

Multiple Values

Another feature of Common Lisp that I've mentioned in passing--in Chapter 11, when I discussed **GETHASH**--is the ability for a single form to return multiple values. I'll discuss it in greater detail now. It is, however, slightly misplaced in a chapter on special operators since the ability to return multiple values isn't provided by just one or two special operators but is deeply integrated into the language. The operators you'll most often use when dealing with multiple values are macros and functions, not special operators. But it is the case that the basic ability to get at multiple return values is provided by a special operator, **MULTIPLE-VALUE-CALL**, upon which the more commonly used **MULTIPLE-VALUE-BIND** macro is built.

The key thing to understand about multiple values is that returning multiple values is quite different from returning a list--if a form returns multiple values, unless you do something specific to capture the multiple values, all but the *primary value* will be silently discarded. To see the distinction, consider the function **GETHASH**, which returns two values: the value found in the hash table and a boolean that's **NIL** when no value was found. If it returned those two values in a list, every time you called **GETHASH** you'd have to take apart the list to get at the actual value, regardless of whether you cared about the second return value. Suppose you have a hash table, `*h*`, that contains numeric values. If **GETHASH** returned a list, you couldn't write something like this:

```
(+ (gethash 'a *h*) (gethash 'b *h*))
```

because + expects its arguments to be numbers, not lists. But because the multiple value mechanism silently discards the secondary return value when it's not wanted, this form works fine.

There are two aspects to using multiple values--returning multiple values and getting at the nonprimary values returned by forms that return multiple values. The starting points for returning multiple values are the functions **VALUES** and **VALUES-LIST**. These are regular functions, not special operators, so their arguments are passed in the normal way. **VALUES** takes a variable number of arguments and returns them as multiple values; **VALUES-LIST** takes a single list and returns its elements as multiple values. In other words:

```
(values-list x) === (apply #'values x)
```

The mechanism by which multiple values are returned is implementation dependent just like the mechanism for passing arguments into functions is. Almost all language constructs that return the value of some subform will "pass through" multiple values, returning all the values returned by the subform. Thus, a function that returns the result of calling **VALUES** or **VALUES-LIST** will itself return multiple values--and so will another function whose result comes from calling the first function. And so on.[11]

But when a form is evaluated in a value position, only the primary value will be used, which is why the previous addition form works the way you'd expect. The special operator **MULTIPLE-VALUE-CALL** provides the mechanism for getting your hands on the multiple values returned by a form. **MULTIPLE-VALUE-CALL** is similar to **FUNCALL** except that while **FUNCALL** is a regular function and, therefore, can

see and pass on only the primary values passed to it, **MULTIPLE-VALUE-CALL** passes, to the function returned by its first subform, *all* the values returned by the remaining subforms.

```
(funcall #'+ (values 1 2) (values 3 4))             ==> 4
(multiple-value-call #'+ (values 1 2) (values 3 4)) ==> 10
```

However, it's fairly rare that you'll simply want to pass all the values returned by a function onto another function. More likely, you'll want to stash the multiple values in different variables and then do something with them. The **MULTIPLE-VALUE-BIND** macro, which you saw in Chapter 11, is the most frequently used operator for accepting multiple return values. Its skeleton looks like this:

```
(multiple-value-bind (variable*) values-form
  body-form*)
```

The *values-form* is evaluated, and the multiple values it returns are bound to the *variables*. Then the *body-forms* are evaluated with those bindings in effect. Thus:

```
(multiple-value-bind (x y) (values 1 2)
  (+ x y)) ==> 3
```

Another macro, **MULTIPLE-VALUE-LIST**, is even simpler--it takes a single form, evaluates it, and collects the resulting multiple values into a list. In other words, it's the inverse of **VALUES-LIST**.

```
CL-USER> (multiple-value-list (values 1 2))
(1 2)
CL-USER> (values-list (multiple-value-list (values 1 2)))
1
2
```

However, if you find yourself using **MULTIPLE-VALUE-LIST** a lot, it may be a sign that some function should be returning a list to start with rather than multiple values.

Finally, if you want to assign multiple values returned by a form to existing variables, you can use **VALUES** as a **SETF**able place. For example:

```
CL-USER> (defparameter *x* nil)
*X*
CL-USER> (defparameter *y* nil)
*Y*
CL-USER> (setf (values *x* *y*) (floor (/ 57 34)))
1
23/34
CL-USER> *x*
1
CL-USER> *y*
23/34
```

EVAL-WHEN

A special operator you'll need to understand in order to write certain kinds of macros is **EVAL-WHEN**. For some reason, Lisp books often treat **EVAL-WHEN** as a wizards-only topic. But the only prerequisite to understanding **EVAL-WHEN** is an understanding of how the two functions **LOAD** and **COMPILE-FILE** interact. And understanding **EVAL-WHEN** will be important as you start writing certain kinds of more sophisticated macros, such as the ones you'll write in Chapters 24 and 31.

I've touched briefly on the relation between **LOAD** and **COMPILE-FILE** in previous chapters, but it's worth reviewing again here. The job of **LOAD** is to load a file and evaluate all the top-level forms it contains. The job of **COMPILE-FILE** is to compile a source file into a FASL file, which can then be loaded with **LOAD** such that `(load "foo.lisp")` and `(load "foo.fasl")` are essentially equivalent.

Because **LOAD** evaluates each form before reading the next, the side effects of evaluating forms earlier in the file can affect how forms later in the form are read and evaluated. For instance, evaluating an **IN-PACKAGE** form changes the value of ***PACKAGE***, which will affect the way subsequent forms are read.[12] Similarly, a **DEFMACRO** form early in a file can define a macro that can then be used by code later in the file.[13]

COMPILE-FILE, on the other hand, normally doesn't evaluate the forms it's compiling; it's when the FASL is loaded that the forms--or their compiled equivalents--will be evaluated. However, **COMPILE-FILE**

must evaluate some forms, such as **IN-PACKAGE** and **DEFMACRO** forms, in order to keep the behavior of `(load "foo.lisp")` and `(load "foo.fasl")` consistent.

So how do macros such as **IN-PACKAGE** and **DEFMACRO** work when processed by **COMPILE-FILE**? In some pre-Common Lisp versions of Lisp, the file compiler simply knew it should evaluate certain macros in addition to compiling them. Common Lisp avoided the need for such kludges by borrowing the **EVAL-WHEN** special operator from Maclisp. This operator, as its name suggests, allows you to control when specific bits of code are evaluated. The skeleton of an **EVAL-WHEN** form looks like this:

```
(eval-when (situation*)
  body-form*)
```

There are three possible *situations*--`:compile-toplevel`, `:load-toplevel`, and `:execute`--and which ones you specify controls when the *body-forms* will be evaluated. An **EVAL-WHEN** with multiple situations is equivalent to several **EVAL-WHEN** forms, one per situation, each with the same body code. To explain the meaning of the three situations, I'll need to explain a bit about how **COMPILE-FILE**, which is also referred to as the *file compiler*, goes about compiling a file.

To explain how **COMPILE-FILE** compiles **EVAL-WHEN** forms, I need to introduce a distinction between compiling *top-level* forms and compiling non-top-level forms. A top-level form is, roughly speaking, one that will be compiled into code that will be run when the FASL is loaded. Thus, all forms that appear directly at the top level of a source file are compiled as top-level forms. Similarly, any forms appearing directly in a top-level **PROGN** are compiled as top-level

forms since the **PROGN** itself doesn't *do* anything--it just groups together its subforms, which will be run when the FASL is loaded.[14] Similarly, forms appearing directly in a **MACROLET** or **SYMBOL-MACROLET** are compiled as top-level forms because after the compiler has expanded the local macros or symbol macros, there will be no remnant of the **MACROLET** or **SYMBOL-MACROLET** in the compiled code. Finally, the expansion of a top-level macro form will be compiled as a top-level form.

Thus, a **DEFUN** appearing at the top level of a source file is a top-level form--the code that defines the function and associates it with its name will run when the FASL is loaded--but the forms within the body of the function, which won't run until the function is called, aren't top-level forms. Most forms are compiled the same when compiled as top-level and non-top-level forms, but the semantics of an **EVAL-WHEN** depend on whether it's being compiled as a top- level form, compiled as a non-top-level form, or simply evaluated, combined with what situations are listed in its situation list.

The situations `:compile-toplevel` and `:load-toplevel` control the meaning of an **EVAL-WHEN** compiled as a top-level form. When `:compile-toplevel` is present, the file compiler will evaluate the subforms at compile time. When `:load-toplevel` is present, it will compile the subforms as top-level forms. If neither of these situations is present in a top-level **EVAL-WHEN**, the compiler ignores it.

When an **EVAL-WHEN** is compiled as a non-top-level form, it's either compiled like a **PROGN**, if the `:execute` situation is specified, or ignored. Similarly, an evaluated **EVAL-WHEN**--which includes top-level

EVAL-WHENs in a source file processed by **LOAD** and **EVAL-WHEN**s evaluated at compile time because they appear as subforms of a top-level **EVAL-WHEN** with the `:compile-toplevel` situation--is treated like a **PROGN** if `:execute` is present and ignored otherwise.

Thus, a macro such as **IN-PACKAGE** can have the necessary effect at both compile time and when loading from source by expanding into an **EVAL-WHEN** like the following:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf *package* (find-package "PACKAGE-NAME")))
```

PACKAGE will be set at compile time because of the `:compile-toplevel` situation, set when the FASL is loaded because of `:load-toplevel`, and set when the source is loaded because of the `:execute`.

There are two ways you're most likely to use **EVAL-WHEN**. One is if you want to write macros that need to save some information at compile time to be used when generating the expansion of other macro forms in the same file. This typically arises with definitional macros where a definition early in a file can affect the code generated for a definition later in the same file. You'll write this kind of macro in Chapter 24.

The other time you might need **EVAL-WHEN** is if you want to put the definition of a macro and helper functions it uses in the same file as code that uses the macro. **DEFMACRO** already includes an **EVAL-WHEN** in its expansion so the macro definition is immediately available to be used later in the file. But **DEFUN** normally doesn't make function definitions available at compile time. But if you use a macro in the same file as it's defined in, you need the macro *and* any functions it

uses to be defined. If you wrap the **DEFUN**s of any helper functions used by the macro in an **EVAL-WHEN** with `:compile-toplevel`, the definitions will be available when the macro's expansion function runs. You'll probably want to include `:load-toplevel` and `:execute` as well since the macros will also need the function definitions after the file is compiled and loaded or if you load the source instead of compiling.

Other Special Operators

The four remaining special operators, **LOCALLY**, **THE**, **LOAD-TIME-VALUE**, and **PROGV**, all allow you to get at parts of the underlying language that can't be accessed any other way. **LOCALLY** and **THE** are part of Common Lisp's declaration system, which is used to communicate things to the compiler that don't affect the meaning of your code but that may help the compiler generate better code--faster, clearer error messages, and so on.[15] I'll discuss declarations briefly in Chapter 32.

The other two, **LOAD-TIME-VALUE** and **PROGV**, are infrequently used, and explaining the reason why you might ever *want* to use them would take longer than explaining what they do. So I'll just tell you what they do so you know they're there. Someday you'll hit on one of those rare times when they're just the thing, and then you'll be ready.

LOAD-TIME-VALUE is used, as its name suggests, to create a value that's determined at load time. When the file compiler compiles code that contains a **LOAD-TIME-VALUE** form, it arranges to evaluate the first subform once, when the FASL is loaded, and for the code containing the **LOAD-TIME-VALUE** form to refer to that value. In other words, instead of writing this:

```
(defvar *loaded-at* (get-universal-time))

(defun when-loaded () *loaded-at*)
```

you can write the following:

```
(defun when-loaded () (load-time-value (get-universal-time)))
```

In code not processed by COMPILE-FILE, LOAD-TIME-VALUE is evaluated once when the code is compiled, which may be when you explicitly compile a function with COMPILE or earlier because of implicit compilation performed by the implementation in the course of evaluating the code. In uncompiled code, LOAD-TIME-VALUE evaluates its form each time it's evaluated.

Finally, PROGV creates new dynamic bindings for variables whose names are determined at runtime. This is mostly useful for implementing embedded interpreters for languages with dynamically scoped variables. The basic skeleton is as follows:

```
(progv symbols-list values-list
  body-form*)
```

where *symbols-list* is a form that evaluates to a list of symbols and *values-list* is a form that evaluates to a list of values. Each symbol is dynamically bound to the corresponding value, and then the *body-forms* are evaluated. The difference between PROGV and LET is that because *symbols-list* is evaluated at runtime, the names of the variables to bind can be determined dynamically. As I say, this isn't something you need to do often.

And that's it for special operators. In the next chapter, I'll get back to hard-nosed practical topics and show you how to use Common Lisp's package system to take control of your namespaces so you can write libraries and applications that can coexist without stomping on each other's names.

---

[1] Of course, if IF wasn't a special operator but some other conditional form, such as COND, was, you could build IF as a macro. Indeed, in many Lisp dialects, starting with McCarthy's

original Lisp, COND was the primitive conditional evaluation operator.

[2]Well, technically those constructs could also expand into a LAMBDA expression since, as I mentioned in Chapter 6, LET could be defined--and was in some earlier Lisps--as a macro that expands into an invocation of an anonymous function.

[3]Surprising as it may seem, it actually is possible to make anonymous functions recurse. However, you must use a rather esoteric mechanism known as the *Y combinator*. But the Y combinator is an interesting theoretical result, not a practical programming tool, so is well outside the scope of this book.

[4]It's not required that WITH-SLOTS be implemented with SYMBOL-MACROLET--in some implementations, WITH-SLOTS may walk the code provided and generate an expansion with `x`, `y`, and `z` already replaced with the appropriate SLOT-VALUE forms. You can see how your implementation does it by evaluating this form:

```
(macroexpand-1 '(with-slots (x y z) obj (list x y z)))
```

However, walking the body is much easier for the Lisp implementation to do than for user code; to replace `x`, `y`, and `z` only when they appear in value positions requires a code walker that understands the syntax of all special operators and that recursively expands all macro forms in order to determine whether their expansions include the symbols in value positions. The Lisp implementation obviously has such a code walker at its disposal, but it's one of the few parts of Lisp that's not exposed to users of the language.

[5]One version of f2cl is available as part of the Common Lisp Open Code Collection (CLOCC): `http://clocc.sourceforge.net/`. By contrast, consider the tricks the authors of f2j, a FORTRAN-to-Java translator, have to play. Although the Java Virtual Machine (JVM) has a goto instruction, it's not directly exposed in Java. So to compile FORTRAN gotos, they first compile the FORTRAN code into legal Java source with calls to a dummy class to represent the labels and gotos. Then they compile the source with a regular Java compiler and postprocess the byte codes to translate the dummy calls into JVM-level byte codes. Clever, but what a pain.

[6]Since this algorithm depends on values returned by RANDOM, you may want to test it with a consistent random seed, which you can get by binding *RANDOM-STATE* to the value of `(make-random-state nil)` around each call to `algorithm-s`. For instance, you can do a basic sanity check of `algorithm-s` by evaluating this:

```
(let ((*random-state* (make-random-state nil))) (algorithm-s 10 200))
```

If your refactorings are all valid, this expression should evaluate to the same list each time.

[7]This is a pretty reasonable restriction--it's not entirely clear what it'd mean to return from a form that has already returned--unless, of course, you're a Scheme programmer. Scheme supports *continuations*, a language construct that makes it possible to return from the same function call more than once. But for a variety of reasons, few, if any, languages other than Scheme support this kind of continuation.

[8]If you're the kind of person who likes to know how things work all the way down to the bits, it may be instructive to think about how you might implement the condition system's macros using **BLOCK**, **TAGBODY**, closures, and dynamic variables.

[9]**UNWIND-PROTECT** is essentially equivalent to `try/finally` constructs in Java and Python.

[10]And indeed, CLSQL, the multi-Lisp, multidatabase SQL interface library, provides a similar macro called `with-database`. CLSQL's home page is at `http://clsql.b9.com`.

[11]A small handful of macros don't pass through extra return values of the forms they evaluate. In particular, the **PROG1** macro, which evaluates a number of forms like a **PROGN** before returning the value of the first form, returns that form's primary value only. Likewise, **PROG2**, which returns the value of the second of its subforms, returns only the primary value. The special operator **MULTIPLE-VALUE-PROG1** is a variant of **PROG1** that returns all the values returned by the first form. It's a minor wart that **PROG1** doesn't already behave like **MULTIPLE-VALUE-PROG1**, but neither is used often enough that it matters much. The **OR** and **COND** macros are also not always transparent to multiple values, returning only the primary value of certain subforms.

[12]The reason loading a file with an **IN-PACKAGE** form in it has no effect on the value of ***PACKAGE*** after **LOAD** returns is because **LOAD** binds ***PACKAGE*** to its current value before doing anything else. In other words, something equivalent to the following **LET** is wrapped around the rest of the code in **LOAD**:

```
(let ((*package* *package*)) ...)
```

Any assignment to ***PACKAGE*** will be to the new binding, and the old binding will be restored when **LOAD** returns. It also binds the variable ***READTABLE***, which I haven't discussed, in the same way.

[13]In some implementations, you may be able to get away with evaluating **DEFUN**s that use undefined macros in the function body as long as the macros are defined before the function is actually called. But that works, if at all, only when **LOAD**ing the definitions from

source, not when compiling with COMPILE-FILE, so in general macro definitions must be evaluated before they're used.

[14]By contrast, the subforms in a top-level LET aren't compiled as top-level forms because they're not run directly when the FASL is loaded. They will run, but it's in the runtime context of the bindings established by the LET. Theoretically, a LET that binds no variables could be treated like a PROGN, but it's not--the forms appearing in a LET are never treated as top-level forms.

[15]The one declaration that has an effect on the semantics of a program is the SPECIAL declaration mentioned in Chapter 6.

Copyright © 2003-2005, Peter Seibel

21. Programming in the Large: Packages and Symbols

In Chapter 4 I discussed how the Lisp reader translates textual names into objects to be passed to the evaluator, representing them with a kind of object called a *symbol*. It turns out that having a built-in data type specifically for representing names is quite handy for a lot of kinds of programming.[1] That, however, isn't the topic of this chapter. In this chapter I'll discuss one of the more immediate and practical aspects of dealing with names: how to avoid name conflicts between independently developed pieces of code.

Suppose, for instance, you're writing a program and decide to use a third-party library. You don't want to have to know the name of every function, variable, class, or macro used in the internals of that library in order to avoid conflicts between those names and the names you use in your program. You'd like for most of the names in the library and the names in your program to be considered distinct even if they happen to have the same textual representation. At the same time, you'd like certain names defined in the library to be readily accessible--the names that make up its public API, which you'll want to use in your program.

In Common Lisp, this namespace problem boils down to a question of controlling how the reader translates textual names into symbols: if you want two occurrences of the same name to be considered the same by the evaluator, you need to make sure the reader uses the

same symbol to represent each name. Conversely, if you want two names to be considered distinct, even if they happen to have the same textual name, you need the reader to create different symbols to represent each name.

How the Reader Uses Packages

In Chapter 4 I discussed briefly how the Lisp reader translates names into symbols, but I glossed over most of the details--now it's time to take a closer look at what actually happens.

I'll start by describing the syntax of names understood by the reader and how that syntax relates to packages. For the moment you can think of a package as a table that maps strings to symbols. As you'll see in the next section, the actual mapping is slightly more flexible than a simple lookup table but not in ways that matter much to the reader. Each package also has a name, which can be used to find the package using the function `FIND-PACKAGE`.

The two key functions that the reader uses to access the name-to-symbol mappings in a package are `FIND-SYMBOL` and `INTERN`. Both these functions take a string and, optionally, a package. If not supplied, the package argument defaults to the value of the global variable `*PACKAGE*`, also called the *current package*.

`FIND-SYMBOL` looks in the package for a symbol with the given string for a name and returns it, or `NIL` if no symbol is found. `INTERN` also will return an existing symbol; otherwise it creates a new symbol with the string as its name and adds it to the package.

Most names you use are *unqualified*, names that contain no colons. When the reader reads such a name, it translates it to a symbol by converting any unescaped letters to uppercase and passing the resulting string to `INTERN`. Thus, each time the reader reads the same

name in the same package, it'll get the same symbol object. This is important because the evaluator uses the object identity of symbols to determine which function, variable, or other program element a given symbol refers to. Thus, the reason an expression such as `(hello-world)` results in calling a particular `hello-world` function is because the reader returns the same symbol when it reads the function call as it did when it read the **DEFUN** form that defined the function.

A name containing either a single colon or a double colon is a package-qualified name. When the reader reads a package-qualified name, it splits the name on the colon(s) and uses the first part as the name of a package and the second part as the name of the symbol. The reader looks up the appropriate package and uses it to translate the symbol name to a symbol object.

A name containing only a single colon must refer to an *external* symbol--one the package *exports* for public use. If the named package doesn't contain a symbol with a given name, or if it does but it hasn't been exported, the reader signals an error. A double-colon name can refer to any symbol from the named package, though it's usually a bad idea--the set of exported symbols defines a package's public interface, and if you don't respect the package author's decision about what names to make public and which ones to keep private, you're asking for trouble down the road. On the other hand, sometimes a package author will neglect to export a symbol that really ought to be public. In that case, a double-colon name lets you get work done without having to wait for the next version of the package to be released.

Two other bits of symbol syntax the reader understands are those for keyword symbols and uninterned symbols. Keyword symbols are written with names starting with a colon. Such symbols are interned in the package named **KEYWORD** and automatically exported. Additionally, when the reader interns a symbol in the **KEYWORD**, it also defines a constant variable with the symbol as both its name and value. This is why you can use keywords in argument lists without quoting them--when they appear in a value position, they evaluate to themselves. Thus:

```
(eql ':foo :foo) ==> T
```

The names of keyword symbols, like all symbols, are converted to all uppercase by the reader before they're interned. The name doesn't include the leading colon.

```
(symbol-name :foo) ==> "FOO"
```

Uninterned symbols are written with a leading `#:`. These names (minus the `#:`) are converted to uppercase as normal and then translated into symbols, but the symbols aren't interned in any package; each time the reader reads a `#:` name, it creates a new symbol. Thus:

```
(eql '#:foo '#:foo) ==> NIL
```

You'll rarely, if ever, write this syntax yourself, but will sometimes see it when you print an s-expression containing symbols returned by the function **GENSYM**.

```
(gensym) ==> #:G3128
```

A Bit of Package and Symbol Vocabulary

As I mentioned previously, the mapping from names to symbols implemented by a package is slightly more flexible than a simple lookup table. At its core, every package contains a name-to-symbol lookup table, but a symbol can be made accessible via an unqualified name in a given package in other ways. To talk sensibly about these other mechanisms, you'll need a little bit of vocabulary.

To start with, all the symbols that can be found in a given package using `FIND-SYMBOL` are said to be *accessible* in that package. In other words, the accessible symbols in a package are those that can be referred to with unqualified names when the package is current.

A symbol can be accessible in two ways. The first is for the package's name-to-symbol table to contain an entry for the symbol, in which case the symbol is said to be *present* in the package. When the reader interns a new symbol in a package, it's added to the package's name-to-symbol table. The package in which a symbol is first interned is called the symbol's *home package*.

The other way a symbol can be accessible in a package is if the package *inherits* it. A package inherits symbols from other packages by *using* the other packages. Only *external* symbols in the used packages are inherited. A symbol is made external in a package by *exporting* it. In addition to causing it to be inherited by using packages, exporting a symbol also--as you saw in the previous section--makes it possible to refer to the symbol using a single-colon qualified name.

To keep the mappings from names to symbols deterministic, the package system allows only one symbol to be accessible in a given package for each name. That is, a package can't have a present symbol and an inherited symbol with the same name or inherit two different symbols, from different packages, with the same name. However, you can resolve conflicts by making one of the accessible symbols a *shadowing* symbol, which makes the other symbols of the same name inaccessible. In addition to its name-to-symbol table, each package maintains a list of shadowing symbols.

An existing symbol can be *imported* into another package by adding it to the package's name-to-symbol table. Thus, the same symbol can be present in multiple packages. Sometimes you'll import symbols simply because you want them to be accessible in the importing package without using their home package. Other times you'll import a symbol because only present symbols can be exported or be shadowing symbols. For instance, if a package needs to use two packages that have external symbols of the same name, one of the symbols must be imported into the using package in order to be added to its shadowing list and make the other symbol inaccessible.

Finally, a present symbol can be *uninterned* from a package, which causes it to be removed from the name-to-symbol table and, if it's a shadowing symbol, from the shadowing list. You might unintern a symbol from a package to resolve a conflict between the symbol and an external symbol from a package you want to use. A symbol that isn't present in any package is called an *uninterned* symbol, can no

longer be read by the reader, and will be printed using the `#:foo` syntax.

Three Standard Packages

In the next section I'll show you how to define your own packages, including how to make one package use another and how to export, shadow, and import symbols. But first let's look at a few packages you've been using already. When you first start Lisp, the value of **`*PACKAGE*`** is typically the `COMMON-LISP-USER` package, also known as `CL-USER`.[2]`CL-USER` uses the package `COMMON-LISP`, which exports all the names defined by the language standard. Thus, when you type an expression at the REPL, all the names of standard functions, macros, variables, and so on, will be translated to the symbols exported from `COMMON-LISP`, and all other names will be interned in the `COMMON-LISP-USER` package. For example, the name **`*PACKAGE*`** is exported from `COMMON-LISP`--if you want to see the value of **`*PACKAGE*`**, you can type this:

```
CL-USER> *package*
#<The COMMON-LISP-USER package>
```

because `COMMON-LISP-USER` uses `COMMON-LISP`. Or you can use a package-qualified name.

```
CL-USER> common-lisp:*package*
#<The COMMON-LISP-USER package>
```

You can even use `COMMON-LISP`'s nickname, `CL`.

```
CL-USER> cl:*package*
#<The COMMON-LISP-USER package>
```

But `*X*` isn't a symbol in `COMMON-LISP`, so you if type this:

```
CL-USER> (defvar *x* 10)
*X*
```

the reader reads **DEFVAR** as the symbol from the COMMON-LISP package and *X* as a symbol in COMMON-LISP-USER.

The REPL can't start in the COMMON-LISP package because you're not allowed to intern new symbols in it; COMMON-LISP-USER serves as a "scratch" package where you can create your own names while still having easy access to all the symbols in COMMON-LISP.[3] Typically, all packages you'll define will also use COMMON-LISP, so you don't have to write things like this:

```
(cl:defun (x) (cl:+ x 2))
```

The third standard package is the **KEYWORD** package, the package the Lisp reader uses to intern names starting with colon. Thus, you can also refer to any keyword symbol with an explicit package qualification of `keyword` like this:

```
CL-USER> :a
:A
CL-USER> keyword:a
:A
CL-USER> (eql :a keyword:a)
T
```

Defining Your Own Packages

Working in `COMMON-LISP-USER` is fine for experiments at the REPL, but once you start writing actual programs you'll want to define new packages so different programs loaded into the same Lisp environment don't stomp on each other's names. And when you write libraries that you intend to use in different contexts, you'll want to define separate packages and then export the symbols that make up the libraries' public APIs.

However, before you start defining packages, it's important to understand one thing about what packages do *not* do. Packages don't provide direct control over who can call what function or access what variable. They provide you with basic control over namespaces by controlling how the reader translates textual names into symbol objects, but it isn't until later, in the evaluator, that the symbol is interpreted as the name of a function or variable or whatever else. Thus, it doesn't make sense to talk about exporting a function or a variable from a package. You can export symbols to make certain names easier to refer to, but the package system doesn't allow you to restrict how those names are used.[4]

With that in mind, you can start looking at how to define packages and tie them together. You define new packages with the macro **`DEFPACKAGE`**, which allows you to not only create the package but to specify what packages it uses, what symbols it exports, and what symbols it imports from other packages and to resolve conflicts by creating shadowing symbols.[5]

I'll describe the various options in terms of how you might use packages while writing a program that organizes e-mail messages into a searchable database. The program is purely hypothetical, as are the libraries I'll refer to--the point is to look at how the packages used in such a program might be structured.

The first package you'd need is one to provide a namespace for the application--you want to be able to name your functions, variables, and so on, without having to worry about name collisions with unrelated code. So you'd define a new package with **DEFPACKAGE**.

If the application is simple enough to be written with no libraries beyond the facilities provided by the language itself, you could define a simple package like this:

```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp))
```

This defines a package, named COM.GIGAMONKEYS.EMAIL-DB, that inherits all the symbols exported by the COMMON-LISP package.[6]

You actually have several choices of how to represent the names of packages and, as you'll see, the names of symbols in a **DEFPACKAGE**. Packages and symbols are named with strings. However, in a **DEFPACKAGE** form, you can specify the names of packages and symbols with *string designators*. A string designator is either a string, which designates itself; a symbol, which designates its name; or a character, which designates a one-character string containing just the character. Using keyword symbols, as in the previous **DEFPACKAGE**, is a common style that allows you to write the names in lowercase--the reader will convert the names to uppercase for you.

You could also write the **DEFPACKAGE** with strings, but then you have to write them in all uppercase, because the true names of most symbols and packages are in fact uppercase because of the case conversion performed by the reader.[7]

```
(defpackage "COM.GIGAMONKEYS.EMAIL-DB"
  (:use "COMMON-LISP"))
```

You could also use nonkeyword symbols--the names in **DEFPACKAGE** aren't evaluated--but then the very act of reading the **DEFPACKAGE** form would cause those symbols to be interned in the current package, which at the very least will pollute that namespace and may also cause problems later if you try to use the package.[8]

To read code in this package, you need to make it the current package with the **IN-PACKAGE** macro:

```
(in-package :com.gigamonkeys.email-db)
```

If you type this expression at the REPL, it will change the value of ***PACKAGE***, affecting how the REPL reads subsequent expressions, until you change it with another call to **IN-PACKAGE**. Similarly, if you include an **IN-PACKAGE** in a file that's loaded with **LOAD** or compiled with **COMPILE-FILE**, it will change the package, affecting the way subsequent expressions in the file are read.[9]

With the current package set to the `COM.GIGAMONKEYS.EMAIL-DB` package, other than names inherited from the `COMMON-LISP` package, you can use any name you want for whatever purpose you want. Thus, you could define a new `hello-world` function that could coexist with the `hello-world` function previously defined in `COMMON-LISP-USER`. Here's the behavior of the existing function:

```
CL-USER> (hello-world)
hello, world
NIL
```

Now you can switch to the new package using **IN-PACKAGE**.[10] Notice how the prompt changes--the exact form is determined by the development environment, but in SLIME the default prompt consists of an abbreviated version of the package name.

```
CL-USER> (in-package :com.gigamonkeys.email-db)
#<The COM.GIGAMONKEYS.EMAIL-DB package>
EMAIL-DB>
```

You can define a new `hello-world` in this package:

```
EMAIL-DB> (defun hello-world () (format t "hello from EMAIL-DB package~%"))
HELLO-WORLD
```

And test it, like this:

```
EMAIL-DB> (hello-world)
hello from EMAIL-DB package
NIL
```

Now switch back to `CL-USER`.

```
EMAIL-DB> (in-package :cl-user)
#<The COMMON-LISP-USER package>
CL-USER>
```

And the old function is undisturbed.

```
CL-USER> (hello-world)
hello, world
NIL
```

Packaging Reusable Libraries

While working on the e-mail database, you might write several functions related to storing and retrieving text that don't have anything in particular to do with e-mail. You might realize that those functions could be useful in other programs and decide to repackage them as a library. You should define a new package, but this time you'll export certain names to make them available to other packages.

```
(defpackage :com.gigamonkeys.text-db
  (:use :common-lisp)
  (:export :open-db
           :save
           :store))
```

Again, you use the `COMMON-LISP` package, because you'll need access to standard functions within `COM.GIGAMONKEYS.TEXT-DB`. The `:export` clause specifies names that will be external in `COM.GIGAMONKEYS.TEXT-DB` and thus accessible in packages that `:use` it. Therefore, after you've defined this package, you can change the definition of the main application package to the following:

```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp :com.gigamonkeys.text-db))
```

Now code written in `COM.GIGAMONKEYS.EMAIL-DB` can use unqualified names to refer to the exported symbols from both `COMMON-LISP` and `COM.GIGAMONKEYS.TEXT-DB`. All other names will continue to be interned directly in the `COM.GIGAMONKEYS.EMAIL-DB` package.

Importing Individual Names

Now suppose you find a third-party library of functions for manipulating e-mail messages. The names used in the library's API are exported from the package `COM.ACME.EMAIL`, so you could `:use` that package to get easy access to those names. But suppose you need to use only one function from this library, and other exported symbols conflict with names you already use (or plan to use) in our own code.[11] In this case, you can import the one symbol you need with an `:import-from` clause in the **`DEFPACKAGE`**. For instance, if the name of the function you want to use is `parse-email-address`, you can change the **`DEFPACKAGE`** to this:

```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp :com.gigamonkeys.text-db)
  (:import-from :com.acme.email :parse-email-address))
```

Now anywhere the name `parse-email-address` appears in code read in the `COM.GIGAMONKEYS.EMAIL-DB` package, it will be read as the symbol from `COM.ACME.EMAIL`. If you need to import more than one symbol from a single package, you can include multiple names after the package name in a single `:import-from` clause. A **`DEFPACKAGE`** can also include multiple `:import-from` clauses in order to import symbols from different packages.

Occasionally you'll run into the opposite situation--a package may export a bunch of names you want to use and a few you don't. Rather than listing all the symbols you *do* want to use in an `:import-from` clause, you can instead `:use` the package and then list the names you *don't* want to inherit in a `:shadow` clause. For instance,

suppose the COM.ACME.TEXT package exports a bunch of names of functions and classes used in text processing. Further suppose that most of these functions and classes are ones you'll want to use in your code, but one of the names, build-index, conflicts with a name you've already used. You can make the build-index from COM.ACME.TEXT inaccessible by shadowing it.

```
(defpackage :com.gigamonkeys.email-db
  (:use
   :common-lisp
   :com.gigamonkeys.text-db
   :com.acme.text)
  (:import-from :com.acme.email :parse-email-address)
  (:shadow :build-index))
```

The :shadow clause causes a new symbol named BUILD-INDEX to be created and added directly to COM.GIGAMONKEYS.EMAIL-DB's name-to-symbol map. Now if the reader reads the name BUILD-INDEX, it will translate it to the symbol in COM.GIGAMONKEYS.EMAIL-DB's map, rather than the one that would otherwise be inherited from COM.ACME.TEXT. The new symbol is also added to a *shadowing symbols list* that's part of the COM.GIGAMONKEYS.EMAIL-DB package, so if you later use another package that also exports a BUILD-INDEX symbol, the package system will know there's no conflict--that you want the symbol from COM.GIGAMONKEYS.EMAIL-DB to be used rather than any other symbols with the same name inherited from other packages.

A similar situation can arise if you want to use two packages that export the same name. In this case the reader won't know which inherited name to use when it reads the textual name. In such situations you must resolve the ambiguity by shadowing the conflicting names. If you don't need to use the name from either package, you could shadow the name with a :shadow clause,

creating a new symbol with the same name in your package. But if you actually want to use one of the inherited symbols, then you need to resolve the ambiguity with a `:shadowing-import-from` clause. Like an `:import-from` clause, a `:shadowing-import-from` clause consists of a package name followed by the names to import from that package. For instance, if `COM.ACME.TEXT` exports a name `SAVE` that conflicts with the name exported from `COM.GIGAMONKEYS.TEXT-DB`, you could resolve the ambiguity with the following **`DEFPACKAGE`**:

```
(defpackage :com.gigamonkeys.email-db
  (:use
   :common-lisp
   :com.gigamonkeys.text-db
   :com.acme.text)
  (:import-from :com.acme.email :parse-email-address)
  (:shadow :build-index)
  (:shadowing-import-from :com.gigamonkeys.text-db :save))
```

Packaging Mechanics

That covers the basics of how to use packages to manage namespaces in several common situations. However, another level of how to use packages is worth discussing--the raw mechanics of how to organize code that uses different packages. In this section I'll discuss a few rules of thumb about how to organize code--where to put your **DEFPACKAGE** forms relative to the code that uses your packages via **IN-PACKAGE**.

Because packages are used by the reader, a package must be defined before you can **LOAD** or **COMPILE-FILE** a file that contains an **IN-PACKAGE** expression switching to that package. Packages also must be defined before other **DEFPACKAGE** forms can refer to them. For instance, if you're going to `:use COM.GIGAMONKEYS.TEXT-DB` in `COM.GIGAMONKEYS.EMAIL-DB`, then `COM.GIGAMONKEYS.TEXT-DB`'s **DEFPACKAGE** must be evaluated before the **DEFPACKAGE** of `COM.GIGAMONKEYS.EMAIL-DB`.

The best first step toward making sure packages exist when they need to is to put all your **DEFPACKAGE**s in files separate from the code that needs to be read in those packages. Some folks like to create a `foo-package.lisp` file for each individual package, and others create a single `packages.lisp` that contains all the **DEFPACKAGE** forms for a group of related packages. Either approach is reasonable, though the one-file-per-package approach also requires that you arrange to load the individual files in the right order according to the interpackage dependencies.

Either way, once all the DEFPACKAGE forms have been separated from the code that will be read in the packages they define, you can arrange to LOAD the files containing the DEFPACKAGEs before you compile or load any of the other files. For simple programs you can do this by hand: simply LOAD the file or files containing the DEFPACKAGE forms, possibly compiling them with COMPILE-FILE first. Then LOAD the files that use those packages, again optionally compiling them first with COMPILE-FILE. Note, however, that the packages don't exist until you LOAD the package definitions, either the source or the files produced by COMPILE-FILE. Thus, if you're compiling everything, you must still LOAD all the package definitions before you can COMPILE-FILE any files to be read in the packages.

Doing these steps by hand will get tedious after a while. For simple programs you can automate the steps by writing a file, `load.lisp`, that contains the appropriate LOAD and COMPILE-FILE calls in the right order. Then you can just LOAD that file. For more complex programs you'll want to use a *system definition* facility to manage loading and compiling files in the right order.[12]

The other key rule of thumb is that each file should contain exactly one IN-PACKAGE form, and it should be the first form in the file other than comments. Files containing DEFPACKAGE forms should start with `(in-package "COMMON-LISP-USER")`, and all other files should contain an IN-PACKAGE of one of your packages.

If you violate this rule and switch packages in the middle of a file, you'll confuse human readers who don't notice the second IN-PACKAGE. Also, many Lisp development environments, particularly

Emacs-based ones such as SLIME, look for an `IN-PACKAGE` to determine the package they should use when communicating with Common Lisp. Multiple `IN-PACKAGE` forms per file may confuse these tools as well.

On the other hand, it's fine to have multiple files read in the same package, each with an identical `IN-PACKAGE` form. It's just a matter of how you like to organize your code.

The other bit of packaging mechanics has to do with how to name packages. Package names live in a flat namespace--package names are just strings, and different packages must have textually distinct names. Thus, you have to consider the possibility of conflicts between package names. If you're using only packages you developed yourself, then you can probably get away with using short names for your packages. But if you're planning to use third-party libraries or to publish your code for use by other programmers, then you need to follow a naming convention that will minimize the possibility of name collisions between different packages. Many Lispers these days are adopting Java-style names, like the ones used in this chapter, consisting of a reversed Internet domain name followed by a dot and a descriptive string.

Package Gotchas

Once you're familiar with packages, you won't spend a bunch of time thinking about them. There's just not that much to them. However, a couple of gotchas that bite most new Lisp programmers make the package system seem more complicated and unfriendly than it really is.

The number-one gotcha arises most commonly when playing around at the REPL. You'll be looking at some library that defines certain interesting functions. You'll try to call one of the functions like this:

```
CL-USER> (foo)
```

and get dropped into the debugger with this error:

```
attempt to call `FOO' which is an undefined function.
   [Condition of type UNDEFINED-FUNCTION]

Restarts:
  0: [TRY-AGAIN] Try calling FOO again.
  1: [RETURN-VALUE] Return a value instead of calling FOO.
  2: [USE-VALUE] Try calling a function other than FOO.
  3: [STORE-VALUE] Setf the symbol-function of FOO and call it again.
  4: [ABORT] Abort handling SLIME request.
  5: [ABORT] Abort entirely from this (lisp) process.
```

Ah, of course--you forgot to use the library's package. So you quit the debugger and try to **`USE-PACKAGE`** the library's package in order to get access to the name `FOO` so you can call the function.

```
CL-USER> (use-package :foolib)
```

But that drops you back into the debugger with this error message:

```
Using package `FOOLIB' results in name conflicts for these symbols: FOO
   [Condition of type PACKAGE-ERROR]

Restarts:
  0: [CONTINUE] Unintern the conflicting symbols from the `COMMON-LISP-USER' package.
  1: [ABORT] Abort handling SLIME request.
  2: [ABORT] Abort entirely from this (lisp) process.
```

Huh? The problem is the first time you called `foo`, the reader read the name `foo` and interned it in `CL-USER` before the evaluator got hold of it and discovered that this newly interned symbol isn't the name of a function. This new symbol then conflicts with the symbol of the same name exported from the `FOOLIB` package. If you had remembered to **`USE-PACKAGE`** `FOOLIB` before you tried to call `foo`, the reader would have read `foo` as the inherited symbol and not interned a `foo` symbol in `CL-USER`.

However, all isn't lost, because the first restart offered by the debugger will patch things up in just the right way: it will unintern the `foo` symbol from `COMMON-LISP-USER`, putting the `CL-USER` package back to the state it was in before you called `foo`, allowing the **`USE-PACKAGE`** to proceed and allowing for the inherited `foo` to be available in `CL-USER`.

This kind of problem can also occur when loading and compiling files. For instance, if you defined a package, `MY-APP`, for code that was going to use functions with names from the `FOOLIB` package, but forgot to `:use FOOLIB`, when you compile the files with an `(in-package :my-app)` in them, the reader will intern new symbols in `MY-APP` for the names that were supposed to be read as symbols from `FOOLIB`. When you try to run the compiled code, you'll get undefined function errors. If you then try to redefine the `MY-APP` package to `:use FOOLIB`, you'll get the conflicting symbols error. The solution is the same: select the restart to unintern the conflicting symbols from `MY-APP`. You'll then need to recompile the code in the `MY-APP` package so it will refer to the inherited names.

The next gotcha is essentially the reverse of the first gotcha. In this case, you'd have defined a package--again, let's say it's `MY-APP`--that

uses another package, say, `FOOLIB`. Now you start writing code in the `MY-APP` package. Although you used `FOOLIB` in order to be able to refer to the `foo` function, `FOOLIB` may export other symbols as well. If you use one of those exported symbols--say, `bar`--as the name of a function in your own code, Lisp won't complain. Instead, the name of your function will be the symbol exported by `FOOLIB`, which will clobber the definition of `bar` from `FOOLIB`.

This gotcha is more insidious because it doesn't cause an error--from the evaluator's point of view it's just being asked to associate a new function with an old name, something that's perfectly legal. It's suspect only because the code doing the redefining was read with a different value for **`*PACKAGE*`** than the name's package. But the evaluator doesn't necessarily know that. However, in most Lisps you'll get an warning about "`redefining BAR, originally defined in`?". You should heed those warnings. If you clobber a definition from a library, you can restore it by reloading the library code with **`LOAD`**.[13]

The last package-related gotcha is, by comparison, quite trivial, but it bites most Lisp programmers at least a few times: you define a package that uses `COMMON-LISP` and maybe a few libraries. Then at the REPL you change to that package to play around. Then you decide to quit Lisp altogether and try to call `(quit)`. However, `quit` isn't a name from the `COMMON-LISP` package--it's defined by the implementation in some implementation-specific package that happens to be used by `COMMON-LISP-USER`. The solution is simple--change packages back to `CL-USER` to quit. Or use the SLIME REPL shortcut `quit`, which will also save you from having to remember that in certain Common Lisp implementations the function to quit is `exit`, not `quit`.

You're almost done with your tour of Common Lisp. In the next chapter I'll discuss the details of the extended `LOOP` macro. After that, the rest of the book is devoted to "practicals": a spam filter, a library for parsing binary files, and various parts of a streaming MP3 server with a Web interface.

---

[1]The kind of programming that relies on a symbol data type is called, appropriately enough, *symbolic* computation. It's typically contrasted to *numeric* programming. An example of a primarily symbolic program that all programmers should be familiar with is a compiler--it treats the text of a program as symbolic data and translates it into a new form.

[2]Every package has one official name and zero or more *nicknames* that can be used anywhere you need to use the package name, such as in package-qualified names or to refer to the package in a `DEFPACKAGE` or `IN-PACKAGE` form.

[3]`COMMON-LISP-USER` is also allowed to provide access to symbols exported by other implementation-defined packages. While this is intended as a convenience for the user--it makes implementation-specific functionality readily accessible--it can also cause confusion for new Lispers: Lisp will complain about an attempt to redefine some name that isn't listed in the language standard. To see what packages `COMMON-LISP-USER` inherits symbols from in a particular implementation, evaluate this expression at the REPL:

```
(mapcar #'package-name (package-use-list :cl-user))
```

And to find out what package a symbol came from originally, evaluate this:

```
(package-name (symbol-package 'some-symbol))
```

with `some-symbol` replaced by the symbol in question. For instance:

```
(package-name (symbol-package 'car)) ==> "COMMON-LISP"
(package-name (symbol-package 'foo)) ==> "COMMON-LISP-USER"
```

Symbols inherited from implementation-defined packages will return some other value.

[4]This is different from the Java package system, which provides a namespace for classes but is also involved in Java's access control mechanism. The non-Lisp language with a package system most like Common Lisp's packages is Perl.

[5]All the manipulations performed by `DEFPACKAGE` can also be performed with functions that man- ipulate package objects. However, since a package generally needs to be fully defined

before it can be used, those functions are rarely used. Also, **DEFPACKAGE** takes care of performing all the package manipulations in the right order--for instance, **DEFPACKAGE** adds symbols to the shadowing list before it tries to use the used packages.

[6]In many Lisp implementations the `:use` clause is optional if you want only to `:use COMMON-LISP`--if it's omitted, the package will automatically inherit names from an implementation-defined list of packages that will usually include `COMMON-LISP`. However, your code will be more portable if you always explicitly specify the packages you want to `:use`. Those who are averse to typing can use the package's nickname and write `(:use :cl)`.

[7]Using keywords instead of strings has another advantage--Allegro provides a "modern mode" Lisp in which the reader does no case conversion of names and in which, instead of a **COMMON-LISP** package with uppercase names, provides a `common-lisp` package with lowercase names. Strictly speaking, this Lisp isn't a conforming Common Lisp since all the names in the standard are defined to be uppercase. But if you write your **DEFPACKAGE** forms using keyword symbols, they will work both in Common Lisp and in this near relative.

[8]Some folks, instead of keywords, use uninterned symbols, using the `#:` syntax.

```
(defpackage #:com.gigamonkeys.email-db
  (:use #:common-lisp))
```

This saves a tiny bit of memory by not interning any symbols in the keyword package--the symbol can become garbage after **DEFPACKAGE** (or the code it expands into) is done with it. However, the difference is so slight that it really boils down to a matter of aesthetics.

[9]The reason to use **IN-PACKAGE** instead of just **SETF**ing ***PACKAGE*** is that **IN-PACKAGE** expands into code that will run when the file is compiled by **COMPILE-FILE** as well as when the file is loaded, changing the way the reader reads the rest of the file during compilation.

[10]In the REPL buffer in SLIME you can also change packages with a REPL shortcut. Type a comma, and then enter `change-package` at the `Command:` prompt.

[11]During development, if you try to `:use` a package that exports a symbol with the same name as a symbol already interned in the using package, Lisp will signal an error and typically offer you a restart that will unintern the offending symbol from the using package. For more on this, see the section "Package Gotchas."

[12]The code for the "Practical" chapters, available from this book's Web site, uses the ASDF system definition library. ASDF stands for Another System Definition Facility.

[13]Some Common Lisp implementations, such as Allegro and SBCL, provide a facility for "locking" the symbols in a particular package so they can be used in defining forms such as **DEFUN**, **DEFVAR**, and **DEFCLASS** only when their home package is the current package.

Copyright © 2003-2005, Peter Seibel

22. LOOP for Black Belts

In Chapter 7 I briefly discussed the extended LOOP macro. As I mentioned then, LOOP provides what is essentially a special-purpose language just for writing iteration constructs.

This might seem like a lot of bother--inventing a whole language just for writing loops. But if you think about the ways loops are used in programs, it actually makes a fair bit of sense. Any program of any size at all will contain quite a number of loops. And while they won't all be the same, they won't all be unique either; patterns will emerge, particularly if you include the code immediately preceding and following the loops--patterns of how things are set up for the loop, patterns in what gets done in the loop proper, and patterns in what gets done after the loop. The LOOP language captures these patterns so you can express them directly.

The LOOP macro has a lot of parts--one of the main complaints of LOOP's detractors is that it's too complex. In this chapter, I'll tackle LOOP head on, giving you a systematic tour of the various parts and how they fit together.

The Parts of a LOOP

You can do the following in a `LOOP`:

- Step variables numerically and over various data structures
- Collect, count, sum, minimize, and maximize values seen while looping
- Execute arbitrary Lisp expressions
- Decide when to terminate the loop
- Conditionally do any of these

Additionally, `LOOP` provides syntax for the following:

- Creating local variables for use within the loop
- Specifying arbitrary Lisp expressions to run before and after the loop proper

The basic structure of a `LOOP` is a set of clauses, each of which begins with a *loop keyword*.[1] How each clause is parsed by the `LOOP` macro depends on the keyword. Some of the main keywords, which you saw in Chapter 7, are `for`, `collecting`, `summing`, `counting`, `do`, and `finally`.

Iteration Control

Most of the so-called iteration control clauses start with the loop keyword `for`, or its synonym `as`,[2] followed by the name of a variable. What follows after the variable name depends on the type of `for` clause.

The subclauses of a `for` clause can iterate over the following:

- Ranges of numbers, up or down, by specified intervals
- The individual items of a list
- The cons cells that make up a list
- The elements of a vector, including subtypes such as strings and bit vectors
- The pairs of a hash table
- The symbols in a package
- The results of repeatedly evaluating a given form

A single loop can have multiple `for` clauses with each clause naming its own variable. When a loop has multiple `for` clauses, the loop terminates as soon as any `for` clause reaches its end condition. For instance, the following loop:

```
(loop
  for item in list
  for i from 1 to 10
  do (something))
```

will iterate at most ten times but may stop sooner if `list` contains fewer than ten items.

Counting Loops

Arithmetic iteration clauses control the number of times the loop body will be executed by stepping a variable over a range of numbers, executing the body once per step. These clauses consist of from one to three of the following *prepositional phrases* after the `for` (or `as`): the *from where* phrase, the *to where* phrase, and the *by how much* phrase.

The *from where* phrase specifies the initial value of the clause's variable. It consists of one of the prepositions `from`, `downfrom`, or `upfrom` followed by a form, which supplies the initial value (a number).

The *to where* phrase specifies a stopping point for the loop and consists of one of the prepositions `to`, `upto`, `below`, `downto`, or `above` followed by a form, which supplies the stopping point. With `upto` and `downto`, the loop body will be terminated (without executing the body again) when the variable passes the stopping point; with `below` and `above`, it stops one iteration earlier.The *by how much* phrase consists of the prepositions `by` and a form, which must evaluate to a positive number. The variable will be stepped (up or down, as determined by the other phrases) by this amount on each iteration or by one if it's omitted.

You must specify at least one of these prepositional phrases. The defaults are to start at zero, increment the variable by one at each iteration, and go forever or, more likely, until some other clause terminates the loop. You can modify any or all of these defaults by

adding the appropriate prepositional phrases. The only wrinkle is that if you want decremental stepping, there's no default *from where* value, so you must specify one with either `from` or `downfrom`. So, the following:

```
(loop for i upto 10 collect i)
```

collects the first eleven integers (zero to ten), but the behavior of this:

```
(loop for i downto -10 collect i)         ; wrong
```

is undefined. Instead, you need to write this:

```
(loop for i from 0 downto -10 collect i)
```

Also note that because **LOOP** is a macro, which runs at compile time, it has to be able to determine the direction to step the variable based solely on the prepositions--not the values of the forms, which may not be known until runtime. So, the following:

```
(loop for i from 10 to 20 ...)
```

works fine since the default is incremental stepping. But this:

```
(loop for i from 20 to 10 ...)
```

won't know to count down from twenty to ten. Worse yet, it won't give you an error--it will just not execute the loop since `i` is already greater than ten. Instead, you must write this:

```
(loop for i from 20 downto 10 ...)
```

or this:

```
(loop for i downfrom 20 to 10 ...)
```

Finally, if you just want a loop that repeats a certain number of times, you can replace a clause of the following form:

```
for i from 1 to number-form
```

with a `repeat` clause like this:

```
repeat number-form
```

These clauses are identical in effect except the `repeat` clause doesn't create an explicit loop variable.

Looping Over Collections and Packages

The `for` clauses for iterating over lists are much simpler than the arithmetic clauses. They support only two prepositional phrases, `in` and `on`.

A phrase of this form:

```
for var in list-form
```

steps *var* over all the elements of the list produced by evaluating *list-form*.

```
(loop for i in (list 10 20 30 40) collect i) ==> (10 20 30 40)
```

Occasionally this clause is supplemented with a `by` phrase, which specifies a function to use to move down the list. The default is `CDR` but can be any function that takes a list and returns a sublist. For instance, you could collect every other element of a list with a loop like this:

```
(loop for i in (list 10 20 30 40) by #'cddr collect i) ==> (10 30)
```

An `on` prepositional phrase is used to step *var* over the cons cells that make up a list.

```
(loop for x on (list 10 20 30) collect x) ==> ((10 20 30) (20 30) (30))
```

This phrase too can take a `by` preposition:

```
(loop for x on (list 10 20 30 40) by #'cddr collect x) ==> ((10 20 30 40) (30 40))
```

Looping over the elements of a vector (which includes strings and bit vectors) is similar to looping over the elements of a list except the

preposition `across` is used instead of `in`.[3] For instance:

```
(loop for x across "abcd" collect x) ==> (#\a #\b #\c #\d)
```

Iterating over a hash table or package is slightly more complicated because hash tables and packages have different sets of values you might want to iterate over--the keys or values in a hash table and the different kinds of symbols in a package. Both kinds of iteration follow the same pattern. The basic pattern looks like this:

```
(loop for var being the things in hash-or-package ...)
```

For hash tables, the possible values for *things* are `hash-keys` and `hash-values`, which cause `var` to be bound to successive values of either the keys or the values of the hash table. The *hash-or-package* form is evaluated once to produce a value, which must be a hash table.

To iterate over a package, *things* can be `symbols`, `present-symbols`, and `external-symbols`, which cause *var* to be bound to each of the symbols accessible in a package, each of the symbols present in a package (in other words, interned or imported into that package), or each of the symbols that have been exported from the package. The *hash-or-package* form is evaluated to produce the name of a package, which is looked up as if by **`FIND-PACKAGE`** or a package object. Synonyms are also available for parts of the `for` clause. In place of `the`, you can use `each`; you can use `of` instead of `in`; and you can write the *things* in the singular (for example, `hash-key` or `symbol`).

Finally, since you'll often want both the keys and the values when iterating over a hash table, the hash table clauses support a `using`

subclause at the end of the hash table clause.

```
(loop for k being the hash-keys in h using (hash-value v) ...)
(loop for v being the hash-values in h using (hash-key k) ...)
```

Both of these loops will bind `k` to each key in the hash table and `v` to the corresponding value. Note that the first element of the `using` subclause must be in the singular form.[4]

Equals-Then Iteration

If none of the other `for` clauses supports exactly the form of variable stepping you need, you can take complete control over stepping with an *equals-then* clause. This clause is similar to the binding clauses in a `DO` loop but cast in a more Algolish syntax. The template is as follows:

```
(loop for var = initial-value-form [ then step-form ] ...)
```

As usual, *var* is the name of the variable to be stepped. Its initial value is obtained by evaluating *initial-value-form* once before the first iteration. In each subsequent iteration, *step-form* is evaluated, and its value becomes the new value of *var*. With no `then` part to the clause, the *initial-value-form* is reevaluated on each iteration to provide the new value. Note that this is different from a `DO` binding clause with no step form.

The *step-form* can refer to other loop variables, including variables created by other `for` clauses later in the loop. For instance:

```
(loop repeat 5
      for x = 0 then y
      for y = 1 then (+ x y)
      collect y) ==> (1 2 4 8 16)
```

However, note that each `for` clause is evaluated separately in the order it appears. So in the previous loop, on the second iteration `x` is set to the value of `y` before `y` changes (in other words, `1`). But `y` is then set to the sum of its old value (still `1`) and the new value of `x`. If the order of the `for` clauses is reversed, the results change.

```
(loop repeat 5
      for y = 1 then (+ x y)
```

```
      for x = 0 then y
      collect y) ==> (1 1 2 4 8)
```

Often, however, you'll want the step forms for multiple variables to be evaluated before any of the variables is given its new value (similar to how DO steps its variables). In that case, you can join multiple `for` clauses by replacing all but the first `for` with `and`. You saw this formulation already in the LOOP version of the Fibonacci computation in Chapter 7. Here's another variant, based on the two previous examples:

```
(loop repeat 5
      for x = 0 then y
      and y = 1 then (+ x y)
      collect y) ==> (1 1 2 3 5)
```

Local Variables

While the main variables needed within a loop are usually declared implicitly in `for` clauses, sometimes you'll need auxiliary variables, which you can declare with `with` clauses.

```
with var [ = value-form ]
```

The name *var* becomes the name of a local variable that will cease to exist when the loop finishes. If the `with` clause contains an `= value-form` part, the variable will be initialized, before the first iteration of the loop, to the value of *value-form*.

Multiple `with` clauses can appear in a loop; each clause is evaluated independently in the order it appears and the value is assigned before proceeding to the next clause, allowing later variables to depend on the value of already declared variables. Mutually independent variables can be declared in one `with` clause with an `and` between each declaration.

Destructuring Variables

One handy feature of **LOOP** I haven't mentioned yet is the ability to destructure list values assigned to loop variables. This lets you take apart the value of lists that would otherwise be assigned to a loop variable, similar to the way **DESTRUCTURING-BIND** works but a bit less elaborate. Basically, you can replace any loop variable in a `for` or `with` clause with a tree of symbols, and the list value that would have been assigned to the simple variable will instead be destructured into variables named by the symbols in the tree. A simple example looks like this:

```
CL-USER> (loop for (a b) in '((1 2) (3 4) (5 6))
               do (format t "a: ~a; b: ~a~%" a b))
a: 1; b: 2
a: 3; b: 4
a: 5; b: 6
NIL
```

The tree can also include dotted lists, in which case the name after the dot acts like a **&rest** parameter, being bound to a list containing any remaining elements of the list. This is particularly handy with `for`/`on` loops since the value is always a list. For instance, this **LOOP** (which I used in Chapter 18 to emit a comma-delimited list):

```
(loop for cons on list
    do (format t "~a" (car cons))
    when (cdr cons) do (format t ", "))
```

could also be written like this:

```
(loop for (item . rest) on list
    do (format t "~a" item)
    when rest do (format t ", "))
```

If you want to ignore a value in the destructured list, you can use **`NIL`** in place of a variable name.

```
(loop for (a nil) in '((1 2) (3 4) (5 6)) collect a) ==> (1 3 5)
```

If the destructuring list contains more variables than there are values in the list, the extra variables are set to **`NIL`**, making all the variables essentially like **`&optional`** parameters. There isn't, however, any equivalent to **`&key`** parameters.

Value Accumulation

The value accumulation clauses are perhaps the most powerful part of **LOOP**. While the iteration control clauses provide a concise syntax for expressing the basic mechanics of looping, they aren't dramatically different from the equivalent mechanisms provided by **DO**, **DOLIST**, and **DOTIMES**.

The value accumulation clauses, on the other hand, provide a concise notation for a handful of common loop idioms having to do with accumulating values while looping. Each accumulation clause starts with a verb and follows this pattern:

```
verb form [ into var ]
```

Each time through the loop, an accumulation clause evaluates *form* and saves the value in a manner determined by the *verb*. With an `into` subclause, the value is saved into the variable named by *var*. The variable is local to the loop, as if it'd been declared in a `with` clause. With no `into` subclause, the accumulation clause instead accumulates a default value for the whole loop expression.

The possible verbs are `collect`, `append`, `nconc`, `count`, `sum`, `maximize`, and `minimize`. Also available as synonyms are the present participle forms: `collecting`, `appending`, `nconcing`, `counting`, `summing`, `maximizing`, and `minimizing`.

A `collect` clause builds up a list containing all the values of *form* in the order they're seen. This is a particularly useful construct because the code you'd have to write to collect a list in order as efficiently as

LOOP does is more painful than you'd normally write by hand.[5] Related to `collect` are the verbs `append` and `nconc`. These verbs also accumulate values into a list, but they join the values, which must be lists, into a single list as if by the functions **APPEND** or **NCONC**. [6]

The remaining accumulation clauses are used to accumulate numeric values. The verb `count` counts the number of times *form* is true, `sum` collects a running total of the values of *form*, `maximize` collects the largest value seen for *form*, and `minimize` collects the smallest. For instance, suppose you define a variable `*random*` that contains a list of random numbers.

```
(defparameter *random* (loop repeat 100 collect (random 10000)))
```

Then the following loop will return a list containing various summary information about the numbers:

```
(loop for i in *random*
   counting (evenp i) into evens
   counting (oddp i) into odds
   summing i into total
   maximizing i into max
   minimizing i into min
   finally (return (list min max total evens odds)))
```

Unconditional Execution

As useful as the value accumulation constructs are, **LOOP** wouldn't be a very good general-purpose iteration facility if there wasn't a way to execute arbitrary Lisp code in the loop body.

The simplest way to execute arbitrary code within a loop body is with a `do` clause. Compared to the clauses I've described so far, with their prepositions and subclauses, `do` is a model of Yodaesque simplicity.[7] A `do` clause consists of the word `do` (or `doing`) followed by one or more Lisp forms that are all evaluated when the `do` clause is. The `do` clause ends at the closing parenthesis of the loop or the next loop keyword.

For instance, to print the numbers from one to ten, you could write this:

```
(loop for i from 1 to 10 do (print i))
```

Another, more dramatic, form of immediate execution is a `return` clause. This clause consists of the word `return` followed by a single Lisp form, which is evaluated, with the resulting value immediately returned as the value of the loop.

You can also break out of a loop in a `do` clause using any of Lisp's normal control flow operators, such as **RETURN** and **RETURN-FROM**. Note that a `return` clause always returns from the immediately enclosing **LOOP** expression, while a **RETURN** or **RETURN-FROM** in a `do` clause can return from any enclosing expression. For instance, compare the following:

```
(block outer
  (loop for i from 0 return 100) ; 100 returned from LOOP
  (print "This will print")
  200) ==> 200
```

to this:

```
(block outer
  (loop for i from 0 do (return-from outer 100)) ; 100 returned from BLOCK
  (print "This won't print")
  200) ==> 100
```

The `do` and `return` clauses are collectively called the *unconditional execution* clauses.

Conditional Execution

Because a `do` clause can contain arbitrary Lisp forms, you can use any Lisp expressions you want, including control constructs such as **`IF`** and **`WHEN`**. So, the following is one way to write a loop that prints only the even numbers between one and ten:

```
(loop for i from 1 to 10 do (when (evenp i) (print i)))
```

However, sometimes you'll want conditional control at the level of loop clauses. For instance, suppose you wanted to sum only the even numbers between one and ten using a `summing` clause. You couldn't write such a loop with a `do` clause because there'd be no way to "call" the `sum i` in the middle of a regular Lisp form. In cases like this, you need to use one of **`LOOP`**'s own conditional expressions like this:

```
(loop for i from 1 to 10 when (evenp i) sum i) ==> 30
```

`LOOP` provides three conditional constructs, and they all follow this basic pattern:

```
conditional test-form loop-clause
```

The *conditional* can be `if`, `when`, or `unless`. The *test-form* is any regular Lisp form, and *loop-clause* can be a value accumulation clause (`count`, `collect`, *and so on), an unconditional execution* clause, or another conditional execution clause. Multiple loop clauses can be attached to a single conditional by joining them with `and`.

As an extra bit of syntactic sugar, within the first loop clause, after the test form, you can use the variable `it` to refer to the value returned by the test form. For instance, the following loop collects the non-**NIL** values found in `some-hash` when looking up the keys in `some-list`:

```
(loop for key in some-list when (gethash key some-hash) collect it)
```

A conditional clause is executed each time through the loop. An `if` or `when` clause executes its *loop-clause* if *test-form* evaluates to true. An `unless` reverses the test, executing *loop-clause* only when *test-form* is **NIL**. Unlike their Common Lisp namesakes, **LOOP**'s `if` and `when` are merely synonyms--there's no difference in their behavior.

All three conditional clauses can also take an `else` branch, which is followed by another loop clause or multiple clauses joined by `and`. When conditional clauses are nested, the set of clauses connected to an inner conditional clause can be closed with the word `end`. The `end` is optional when not needed to disambiguate a nested conditional--the end of a conditional clause will be inferred from the end of the loop or the start of another clause not joined by `and`.

The following rather silly loop demonstrates the various forms of **LOOP** conditionals. The `update-analysis` function will be called each time through the loop with the latest values of the various variables accumulated by the clauses within the conditionals.

```
(loop for i from 1 to 100
      if (evenp i)
        minimize i into min-even and
        maximize i into max-even and
        unless (zerop (mod i 4))
          sum i into even-not-fours-total
        end
        and sum i into even-total
```

```
else
  minimize i into min-odd and
  maximize i into max-odd and
  when (zerop (mod i 5))
    sum i into fives-total
  end
  and sum i into odd-total
do (update-analysis min-even
                    max-even
                    min-odd
                    max-odd
                    even-total
                    odd-total
                    fives-total
                    even-not-fours-total))
```

Setting Up and Tearing Down

One of the key insights the designers of the LOOP language had about actual loops "in the wild" is that the loop proper is often preceded by a bit of code to set things up and then followed by some more code that does something with the values computed by the loop. A trivial example, in Perl,[8] might look like this:

```
my $evens_sum = 0;
my $odds_sum  = 0;
foreach my $i (@list_of_numbers) {
  if ($i % 2) {
    $odds_sum += $i;
  } else {
    $evens_sum += $i;
  }
}
if ($evens_sum > $odds_sum) {
  print "Sum of evens greater\n";
} else {
  print "Sum of odds greater\n";
}
```

The loop proper in this code is the `foreach` statement. But the `foreach` loop doesn't stand on its own: the code in the loop body refers to variables declared in the two lines before the loop.[9] And the work the loop does is all for naught without the `if` statement after the loop that actually reports the results. In Common Lisp, of course, the LOOP construct is an expression that returns a value, so there's even more often a need to do something after the loop proper, namely, generate the return value.

So, said the LOOP designers, let's give a way to include the code that's really part of the loop in the loop itself. Thus, LOOP provides two keywords, `initially` and `finally`, that introduce code to be run outside the loop's main body.

After the `initially` or `finally`, these clauses consist of all the Lisp forms up to the start of the next loop clause or the end of the loop. All the `initially` forms are combined into a single *prologue*, which runs once, immediately after all the local loop variables are initialized and before the body of the loop. The `finally` forms are similarly combined into a *epilogue* to be run after the last iteration of the loop body. Both the prologue and epilogue code can refer to local loop variables.

The prologue is always run, even if the loop body iterates zero times. The loop can return without running the epilogue if any of the following happens:

- A `return` clause executes.
- **`RETURN`** , **`RETURN-FROM`**, or another transfer of control construct is called from within a Lisp form within the body.[10]
- The loop is terminated by an `always`, `never`, or `thereis` clause, as I'll discuss in the next section.

Within the epilogue code, **`RETURN`** or **`RETURN-FROM`** can be used to explicitly provide a return value for the loop. Such an explicit return value will take precedence over any value that might otherwise be provided by an accumulation or termination test clause.

To allow **`RETURN-FROM`** to be used to return from a specific loop (useful when nesting **`LOOP`** expressions), you can name a **`LOOP`** with the loop keyword `named`. If a `named` clause appears in a loop, it must be the first clause. For a simple example, assume `lists` is a list of lists and you want to find an item that matches some criteria in one of those nested lists. You could find it with a pair of nested loops like this:

```
(loop named outer for list in lists do
     (loop for item in list do
          (if (what-i-am-looking-for-p item)
            (return-from outer item))))
```

Termination Tests

While the `for` and `repeat` clauses provide the basic infrastructure for controlling the number of iterations, sometimes you'll need to break out of a loop early. You've already seen how a `return` clause or a **`RETURN`** or **`RETURN-FROM`** within a `do` clause can immediately terminate the loop; but just as there are common patterns for accumulating values, there are also common patterns for deciding when it's time to bail on a loop. These patterns are supported in **`LOOP`** by the termination clauses, `while`, `until`, `always`, `never`, and `thereis`. They all follow the same pattern.

```
loop-keyword test-form
```

All five evaluate *test-form* each time through the iteration and decide, based on the resulting value, whether to terminate the loop. They differ in what happens after they terminate the loop--if they do--and how they decide.

The loop keywords `while` and `until` introduce the "mild" termination clauses. When they decide to terminate the loop, control passes to the epilogue, skipping the rest of the loop body. The epilogue can then return a value or do whatever it wants to finish the loop. A `while` clause terminates the loop the first time the test form is false; `until`, conversely, stops it the first time the test form is true.

Another form of mild termination is provided by the **`LOOP-FINISH`** macro. This is a regular Lisp form, not a loop clause, so it can be used anywhere within the Lisp forms of a `do` clause. It also causes an immediate jump to the loop epilogue. It can be useful when the

decision to break out of the loop can't be easily condensed into a single form that can be used with a `while` or `until` clause.

The other three clauses--`always`, `never`, and `thereis`--terminate the loop with extreme prejudice; they immediately return from the loop, skipping not only any subsequent loop clauses but also the epilogue. They also provide a default value for the loop even when they don't cause the loop to terminate. However, if the loop is *not* terminated by one of these termination tests, the epilogue is run and can return a value other than the default provided by the termination clauses.

Because these clauses provide their own return values, they can't be combined with accumulation clauses unless the accumulation clause has an `into` subclause. The compiler (or interpreter) should signal an error at compile time if they are.The `always` and `never` clauses return only boolean values, so they're most useful when you need to use a loop expression as part of a predicate. You can use `always` to check that the test form is true on every iteration of the loop. Conversely, `never` tests that the test form evaluates to **`NIL`** on every iteration. If the test form fails (returning **`NIL`** in an `always` clause or non-**`NIL`** in a `never` clause), the loop is immediately terminated, returning **`NIL`**. If the loop runs to completion, the default value of **`T`** is provided.

For instance, if you want to test that all the numbers in a list, `numbers`, are even, you can write this:

```
(if (loop for n in numbers always (evenp n))
    (print "All numbers even."))
```

Equivalently you could write the following:

```
(if (loop for n in numbers never (oddp n))
    (print "All numbers even."))
```

A `thereis` clause is used to test whether the test form is *ever* true. As soon as the test form returns a non-**`NIL`** value, the loop is terminated, returning that value. If the loop runs to completion, the `thereis` clause provides a default return value of **`NIL`**.

```
(loop for char across "abc123" thereis (digit-char-p char)) ==> 1

(loop for char across "abcdef" thereis (digit-char-p char)) ==> NIL
```

Putting It All Together

Now you've seen all the main features of the LOOP facility. You can combine any of the clauses I've discussed as long as you abide by the following rules:

- The `named` clause, if any, must be the first clause.
- After the `named` clause come all the `initially`, `with`, `for`, and `repeat` clauses.
- Then comes the body clauses: conditional and unconditional execution, accumulation, and termination test.[11]
- End with any `finally` clauses.

The LOOP macro will expand into code that performs the following actions:

- Initializes all local loop variables as declared with `with` or `for` clauses as well as those implicitly created by accumulation clauses. The initial value forms are evaluated in the order the clauses appear in the loop.
- Execute the forms provided by any `initially` clauses--the prologue--in the order they appear in the loop.
- Iterate, executing the body of the loop as described in the next paragraph.
- Execute the forms provided by any `finally` clauses--the epilogue--in the order they appear in the loop.

While the loop is iterating, the body is executed by first stepping any iteration control variables and then executing any conditional or unconditional execution, accumulation, or termination test clauses in the order they appear in the loop code. If any of the clauses in the

loop body terminate the loop, the rest of the body is skipped and the loop returns, possibly after running the epilogue.

And that's pretty much all there is to it.[12] You'll use `LOOP` fairly often in the code later in this book, so it's worth having some knowledge of it. Beyond that, it's up to you how much you use it.

And with that, you're ready to dive into the practical chapters that make up the rest of the book--up first, writing a spam filter.

---

[1]The term *loop keyword* is a bit unfortunate, as loop keywords aren't keywords in the normal sense of being symbols in the `KEYWORD` package. In fact, any symbol, from any package, with the appropriate name will do; the `LOOP` macro cares only about their names. Typically, though, they're written with no package qualifier and are thus read (and interned as necessary) in the current package.

[2]Because one of the goals of `LOOP` is to allow loop expressions to be written with a quasi-English syntax, many of the keywords have synonyms that are treated the same by `LOOP` but allow some freedom to express things in slightly more idiomatic English for different contexts.

[3]You may wonder why `LOOP` can't figure out whether it's looping over a list or a vector without needing different prepositions. This is another consequence of `LOOP` being a macro: the value of the list or vector won't be known until runtime, but `LOOP`, as a macro, has to generate code at compile time. And `LOOP`'s designers wanted it to generate extremely efficient code. To be able to generate efficient code for looping across, say, a vector, it needs to know at compile time that the value will be a vector at runtime--thus, the different prepositions are needed.

[4]Don't ask me why `LOOP`'s authors chickened out on the no-parentheses style for the `using` subclause.

[5]The trick is to keep ahold of the tail of the list and add new cons cells by `SETF`ing the `CDR` of the tail. A handwritten equivalent of the code generated by `(loop for i upto 10 collect i)` would look like this:

```
(do ((list nil) (tail nil) (i 0 (1+ i)))
    ((> i 10) list)
  (let ((new (cons i nil)))
    (if (null list)
        (setf list new)
        (setf (cdr tail) new))
    (setf tail new)))
```

Of course you'll rarely, if ever, write code like that. You'll use either **LOOP** or (if, for some reason, you don't want to use **LOOP**) the standard **PUSH**/**NREVERSE** idiom for collecting values.

[6]Recall that **NCONC** is the destructive version of **APPEND**--it's safe to use an `nconc` clause only if the values you're collecting are fresh lists that don't share any structure with other lists. For instance, this is safe:

```
(loop for i upto 3 nconc (list i i)) ==> (0 0 1 1 2 2 3 3)
```

But this will get you into trouble:

```
(loop for i on (list 1 2 3) nconc i) ==> undefined
```

The later will most likely get into an infinite loop as the various parts of the list produced by (list 1 2 3) are destructively modified to point to each other. But even that's not guaranteed--the behavior is simply undefined.

[7]"No! Try not. Do . . . or do not. There is no try." -- Yoda, *The Empire Strikes Back*

[8]I'm not picking on Perl here--this example would look pretty much the same in any language that bases its syntax on C's.

[9]Perl would let you get away with not declaring those variables if your program didn't `use strict`. But you should *always* `use strict` in Perl. The equivalent code in Python, Java, or C would always require the variables to be declared.

[10]You can cause a loop to finish normally, running the epilogue, from Lisp code executed as part of the loop body with the local macro **LOOP-FINISH**.

[11]Some Common Lisp implementations will let you get away with mixing body clauses and `for` clauses, but that's strictly undefined, and some implementations will reject such loops.

[12]The one aspect of **LOOP** I haven't touched on at all is the syntax for declaring the types of loop variables. Of course, I haven't discussed type declarations outside of **LOOP** either. I'll cover the general topic a bit in Chapter 32. For information on how they work with **LOOP**, consult your favorite Common Lisp reference.

Copyright © 2003-2005, Peter Seibel

23. Practical: A Spam Filter

In 2002 Paul Graham, having some time on his hands after selling Viaweb to Yahoo, wrote the essay "A Plan for Spam"[1] that launched a minor revolution in spam-filtering technology. Prior to Graham's article, most spam filters were written in terms of handcrafted rules: if a message has *XXX* in the subject, it's probably a spam; if a message has a more than three or more words in a row in ALL CAPITAL LETTERS, it's probably a spam. Graham spent several months trying to write such a rule-based filter before realizing it was fundamentally a soul-sucking task.

> To recognize individual spam features you have to try to get into the mind of the spammer, and frankly I want to spend as little time inside the minds of spammers as possible.

To avoid having to think like a spammer, Graham decided to try distinguishing spam from nonspam, a.k.a. *ham*, based on statistics gathered about which words occur in which kinds of e-mails. The filter would keep track of how often specific words appear in both spam and ham messages and then use the frequencies associated with the words in a new message to compute a probability that it was either spam or ham. He called his approach *Bayesian* filtering after the statistical technique that he used to combine the individual word frequencies into an overall probability.[2]

The Heart of a Spam Filter

In this chapter, you'll implement the core of a spam-filtering engine. You won't write a soup-to-nuts spam-filtering application; rather, you'll focus on the functions for classifying new messages and training the filter.

This application is going to be large enough that it's worth defining a new package to avoid name conflicts. For instance, in the source code you can download from this book's Web site, I use the package name `COM.GIGAMONKEYS.SPAM`, defining a package that uses both the standard `COMMON-LISP` package and the `COM.GIGAMONKEYS.PATHNAMES` package from Chapter 15, like this:

```
(defpackage :com.gigamonkeys.spam
  (:use :common-lisp :com.gigamonkeys.pathnames))
```

Any file containing code for this application should start with this line:

```
(in-package :com.gigamonkeys.spam)
```

You can use the same package name or replace `com.gigamonkeys` with some domain you control.[3]

You can also type this same form at the REPL to switch to this package to test the functions you write. In SLIME this will change the prompt from `CL-USER>` to `SPAM>` like this:

```
CL-USER> (in-package :com.gigamonkeys.spam)
#<The COM.GIGAMONKEYS.SPAM package>
SPAM>
```

Once you have a package defined, you can start on the actual code. The main function you'll need to implement has a simple job--take

the text of a message as an argument and classify the message as spam, ham, or unsure. You can easily implement this basic function by defining it in terms of other functions that you'll write in a moment.

```
(defun classify (text)
  (classification (score (extract-features text))))
```

Reading from the inside out, the first step in classifying a message is to extract features to pass to the `score` function. In `score` you'll compute a value that can then be translated into one of three classifications--spam, ham, or unsure--by the function `classification`. Of the three functions, `classification` is the simplest. You can assume `score` will return a value near 1 if the message is a spam, near 0 if it's a ham, and near .5 if it's unclear.

Thus, you can implement `classification` like this:

```
(defparameter *max-ham-score* .4)
(defparameter *min-spam-score* .6)

(defun classification (score)
  (cond
    ((<= score *max-ham-score*) 'ham)
    ((>= score *min-spam-score*) 'spam)
    (t 'unsure)))
```

The `extract-features` function is almost as straightforward, though it requires a bit more code. For the moment, the features you'll extract will be the words appearing in the text. For each word, you need to keep track of the number of times it has been seen in a spam and the number of times it has been seen in a ham. A convenient way to keep those pieces of data together with the word itself is to define a class, `word-feature`, with three slots.

```
(defclass word-feature ()
  ((word
    :initarg :word
    :accessor word
```

```
   :initform (error "Must supply :word")
   :documentation "The word this feature represents.")
  (spam-count
   :initarg :spam-count
   :accessor spam-count
   :initform 0
   :documentation "Number of spams we have seen this feature in.")
  (ham-count
   :initarg :ham-count
   :accessor ham-count
   :initform 0
   :documentation "Number of hams we have seen this feature in.")))
```

You'll keep the database of features in a hash table so you can easily find the object representing a given feature. You can define a special variable, `*feature-database*`, to hold a reference to this hash table.

```
(defvar *feature-database* (make-hash-table :test #'equal))
```

You should use **DEFVAR** rather than **DEFPARAMETER** because you don't want `*feature-database*` to be reset if you happen to reload the file containing this definition during development--you might have data stored in `*feature-database*` that you don't want to lose. Of course, that means if you *do* want to clear out the feature database, you can't just reevaluate the **DEFVAR** form. So you should define a function `clear-database`.

```
(defun clear-database ()
  (setf *feature-database* (make-hash-table :test #'equal)))
```

To find the features present in a given message, the code will need to extract the individual words and then look up the corresponding `word-feature` object in `*feature-database*`. If `*feature-database*` contains no such feature, it'll need to create a new `word-feature` to represent the word. You can encapsulate that bit of logic in a function, `intern-feature`, that takes a word and returns the appropriate feature, creating it if necessary.

```
(defun intern-feature (word)
  (or (gethash word *feature-database*)
      (setf (gethash word *feature-database*)
            (make-instance 'word-feature :word word))))
```

You can extract the individual words from the message text using a regular expression. For example, using the Common Lisp Portable Perl-Compatible Regular Expression (CL-PPCRE) library written by Edi Weitz, you can write `extract-words` like this:[4]

```
(defun extract-words (text)
  (delete-duplicates
   (cl-ppcre:all-matches-as-strings "[a-zA-Z]{3,}" text)
   :test #'string=))
```

Now all that remains to implement `extract-features` is to put `extract-features` and `intern-feature` together. Since `extract-words` returns a list of strings and you want a list with each string translated to the corresponding `word-feature`, this is a perfect time to use **`MAPCAR`**.

```
(defun extract-features (text)
  (mapcar #'intern-feature (extract-words text)))
```

You can test these functions at the REPL like this:

```
SPAM> (extract-words "foo bar baz")
("foo" "bar" "baz")
```

And you can make sure the **`DELETE-DUPLICATES`** is working like this:

```
SPAM> (extract-words "foo bar baz foo bar")
("baz" "foo" "bar")
```

You can also test `extract-features`.

```
SPAM> (extract-features "foo bar baz foo bar")
(#<WORD-FEATURE @ #x71ef28da> #<WORD-FEATURE @ #x71e3809a>
 #<WORD-FEATURE @ #x71ef28aa>)
```

However, as you can see, the default method for printing arbitrary objects isn't very informative. As you work on this program, it'll be useful to be able to print `word-feature` objects in a less opaque way. Luckily, as I mentioned in Chapter 17, the printing of all objects is implemented in terms of a generic function **`PRINT-OBJECT`**, so to change the way `word-feature` objects are printed, you just need to define a method on **`PRINT-OBJECT`** that specializes on `word-feature`. To make implementing such methods easier, Common Lisp provides the macro **`PRINT-UNREADABLE-OBJECT`**.[5]

The basic form of **`PRINT-UNREADABLE-OBJECT`** is as follows:

```
(print-unreadable-object (object stream-variable &key type identity)
  body-form*)
```

The *object* argument is an expression that evaluates to the object to be printed. Within the body of **`PRINT-UNREADABLE-OBJECT`**, *stream-variable* is bound to a stream to which you can print anything you want. Whatever you print to that stream will be output by **`PRINT-UNREADABLE-OBJECT`** and enclosed in the standard syntax for unreadable objects, `#<>`.[6]

`PRINT-UNREADABLE-OBJECT` also lets you include the type of the object and an indication of the object's identity via the keyword parameters *type* and *identity*. If they're non-**`NIL`**, the output will start with the name of the object's class and end with an indication of the object's identity similar to what's printed by the default **`PRINT-OBJECT`** method for **`STANDARD-OBJECT`**s. For `word-feature`, you probably want to define a **`PRINT-OBJECT`** method that includes the type but not the identity along with the values of the `word`, `ham-count`, and `spam-count` slots. Such a method would look like this:

```
(defmethod print-object ((object word-feature) stream)
  (print-unreadable-object (object stream :type t)
    (with-slots (word ham-count spam-count) object
      (format stream "~s :hams ~d :spams ~d" word ham-count spam-count))))
```

Now when you test `extract-features` at the REPL, you can see more clearly what features are being extracted.

```
SPAM> (extract-features "foo bar baz foo bar")
(#<WORD-FEATURE "baz" :hams 0 :spams 0>
 #<WORD-FEATURE "foo" :hams 0 :spams 0>
 #<WORD-FEATURE "bar" :hams 0 :spams 0>)
```

Training the Filter

Now that you have a way to keep track of individual features, you're almost ready to implement `score`. But first you need to write the code you'll use to train the spam filter so `score` will have some data to use. You'll define a function, `train`, that takes some text and a symbol indicating what kind of message it is--`ham` or `spam`--and that increments either the ham count or the spam count of all the features present in the text as well as a global count of hams or spams processed. Again, you can take a top-down approach and implement it in terms of other functions that don't yet exist.

```
(defun train (text type)
  (dolist (feature (extract-features text))
    (increment-count feature type))
  (increment-total-count type))
```

You've already written `extract-features`, so next up is `increment-count`, which takes a `word-feature` and a message type and increments the appropriate slot of the feature. Since there's no reason to think that the logic of incrementing these counts is going to change for different kinds of objects, you can write this as a regular function.[7] Because you defined both `ham-count` and `spam-count` with an `:accessor` option, you can use **`INCF`** and the accessor functions created by **`DEFCLASS`** to increment the appropriate slot.

```
(defun increment-count (feature type)
  (ecase type
    (ham (incf (ham-count feature)))
    (spam (incf (spam-count feature)))))
```

The **`ECASE`** construct is a variant of **`CASE`**, both of which are similar to `case` statements in Algol-derived languages (renamed `switch` in C

and its progeny). They both evaluate their first argument--the *key form*--and then find the clause whose first element--the *key*--is the same value according to **EQL**. In this case, that means the variable `type` is evaluated, yielding whatever value was passed as the second argument to `increment-count`.

The keys aren't evaluated. In other words, the value of `type` will be compared to the literal objects read by the Lisp reader as part of the **ECASE** form. In this function, that means the keys are the symbols `ham` and `spam`, not the values of any variables named `ham` and `spam`. So, if `increment-count` is called like this:

```
(increment-count some-feature 'ham)
```

the value of `type` will be the symbol `ham`, and the first branch of the **ECASE** will be evaluated and the feature's ham count incremented. On the other hand, if it's called like this:

```
(increment-count some-feature 'spam)
```

then the second branch will run, incrementing the spam count. Note that the symbols `ham` and `spam` are quoted when calling `increment-count` since otherwise they'd be evaluated as the names of variables. But they're not quoted when they appear in **ECASE** since **ECASE** doesn't evaluate the keys.[8]

The *E* in **ECASE** stands for "exhaustive" or "error," meaning **ECASE** should signal an error if the key value is anything other than one of the keys listed. The regular **CASE** is looser, returning **NIL** if no matching clause is found.

To implement `increment-total-count`, you need to decide where to store the counts; for the moment, two more special variables, `*total-spams*` and `*total-hams*`, will do fine.

```
(defvar *total-spams* 0)
(defvar *total-hams* 0)

(defun increment-total-count (type)
  (ecase type
    (ham (incf *total-hams*))
    (spam (incf *total-spams*))))
```

You should use **`DEFVAR`** to define these two variables for the same reason you used it with `*feature-database*`--they'll hold data built up while you run the program that you don't necessarily want to throw away just because you happen to reload your code during development. But you'll want to reset those variables if you ever reset `*feature-database*`, so you should add a few lines to `clear-database` as shown here:

```
(defun clear-database ()
  (setf
   *feature-database* (make-hash-table :test #'equal)
   *total-spams* 0
   *total-hams* 0))
```

Per-Word Statistics

The heart of a statistical spam filter is, of course, the functions that compute statistics-based probabilities. The mathematical nuances[9] of why exactly these computations work are beyond the scope of this book--interested readers may want to refer to several papers by Gary Robinson.[10] I'll focus rather on how they're implemented.

The starting point for the statistical computations is the set of measured values--the frequencies stored in `*feature-database*`, `*total-spams*`, and `*total-hams*`. Assuming that the set of messages trained on is statistically representative, you can treat the observed frequencies as probabilities of the same features showing up in hams and spams in future messages.

The basic plan is to classify a message by extracting the features it contains, computing the individual probability that a given message containing the feature is a spam, and then combining all the individual probabilities into a total score for the message. Messages with many "spammy" features and few "hammy" features will receive a score near 1, and messages with many hammy features and few spammy features will score near 0.

The first statistical function you need is one that computes the basic probability that a message containing a given feature is a spam. By one point of view, the probability that a given message containing the feature is a spam is the ratio of spam messages containing the feature to all messages containing the feature. Thus, you could compute it this way:

```
(defun spam-probability (feature)
  (with-slots (spam-count ham-count) feature
    (/ spam-count (+ spam-count ham-count))))
```

The problem with the value computed by this function is that it's strongly affected by the overall probability that *any* message will be a spam or a ham. For instance, suppose you get nine times as much ham as spam in general. A completely neutral feature will then appear in one spam for every nine hams, giving you a spam probability of 1/10 according to this function.

But you're more interested in the probability that a given feature will appear in a spam message, independent of the overall probability of getting a spam or ham. Thus, you need to divide the spam count by the total number of spams trained on and the ham count by the total number of hams. To avoid division-by-zero errors, if either of `*total-spams*` or `*total-hams*` is zero, you should treat the corresponding frequency as zero. (Obviously, if the total number of either spams or hams is zero, then the corresponding per-feature count will also be zero, so you can treat the resulting frequency as zero without ill effect.)

```
(defun spam-probability (feature)
  (with-slots (spam-count ham-count) feature
    (let ((spam-frequency (/ spam-count (max 1 *total-spams*)))
          (ham-frequency (/ ham-count (max 1 *total-hams*))))
      (/ spam-frequency (+ spam-frequency ham-frequency)))))
```

This version suffers from another problem--it doesn't take into account the number of messages analyzed to arrive at the per-word probabilities. Suppose you've trained on 2,000 messages, half spam and half ham. Now consider two features that have appeared only in spams. One has appeared in all 1,000 spams, while the other appeared only once. According to the current definition of

`spam-probability`, the appearance of either feature predicts that a message is spam with equal probability, namely, 1.

However, it's still quite possible that the feature that has appeared only once is actually a neutral feature--it's obviously rare in either spams or hams, appearing only once in 2,000 messages. If you trained on another 2,000 messages, it might very well appear one more time, this time in a ham, making it suddenly a neutral feature with a spam probability of .5.

So it seems you might like to compute a probability that somehow factors in the number of data points that go into each feature's probability. In his papers, Robinson suggested a function based on the Bayesian notion of incorporating observed data into prior knowledge or assumptions. Basically, you calculate a new probability by starting with an assumed prior probability and a weight to give that assumed probability before adding new information. Robinson's function is this:

```
(defun bayesian-spam-probability (feature &optional
                                  (assumed-probability 1/2)
                                  (weight 1))
  (let ((basic-probability (spam-probability feature))
        (data-points (+ (spam-count feature) (ham-count feature))))
    (/ (+ (* weight assumed-probability)
          (* data-points basic-probability))
       (+ weight data-points))))
```

Robinson suggests values of 1/2 for `assumed-probability` and 1 for `weight`. Using those values, a feature that has appeared in one spam and no hams has a `bayesian-spam-probability` of 0.75, a feature that has appeared in 10 spams and no hams has a `bayesian-spam-probability` of approximately 0.955, and one that has

matched in 1,000 spams and no hams has a spam probability of approximately 0.9995.

Combining Probabilities

Now that you can compute the `bayesian-spam-probability` of each individual feature you find in a message, the last step in implementing the `score` function is to find a way to combine a bunch of individual probabilities into a single value between 0 and 1.

If the individual feature probabilities were independent, then it'd be mathematically sound to multiply them together to get a combined probability. But it's unlikely they actually are independent--certain features are likely to appear together, while others never do.[11]

Robinson proposed using a method for combining probabilities invented by the statistician R. A. Fisher. Without going into the details of exactly why his technique works, it's this: First you combine the probabilities by multiplying them together. This gives you a number nearer to 0 the more low probabilities there were in the original set. Then take the log of that number and multiply by -2. Fisher showed in 1950 that if the individual probabilities were independent and drawn from a uniform distribution between 0 and 1, then the resulting value would be on a chi-square distribution. This value and twice the number of probabilities can be fed into an inverse chi-square function, and it'll return the probability that reflects the likelihood of obtaining a value that large or larger by combining the same number of randomly selected probabilities. When the inverse chi-square function returns a low probability, it means there was a disproportionate number of low probabilities (either a lot of relatively low probabilities or a few very low probabilities) in the individual probabilities.

To use this probability in determining whether a given message is a spam, you start with a *null hypothesis*, a straw man you hope to knock down. The null hypothesis is that the message being classified is in fact just a random collection of features. If it were, then the individual probabilities--the likelihood that each feature would appear in a spam--would also be random. That is, a random selection of features would usually contain some features with a high probability of appearing in spam and other features with a low probability of appearing in spam. If you were to combine these randomly selected probabilities according to Fisher's method, you should get a middling combined value, which the inverse chi-square function will tell you is quite likely to arise just by chance, as, in fact, it would have. But if the inverse chi-square function returns a very low probability, it means it's unlikely the probabilities that went into the combined value were selected at random; there were too many low probabilities for that to be likely. So you can reject the null hypothesis and instead adopt the alternative hypothesis that the features involved were drawn from a biased sample--one with few high spam probability features and many low spam probability features. In other words, it must be a ham message.

However, the Fisher method isn't symmetrical since the inverse chi-square function returns the probability that a given number of randomly selected probabilities would combine to a value as large or larger than the one you got by combining the actual probabilities. This asymmetry works to your advantage because when you reject the null hypothesis, you know what the more likely hypothesis is. When you combine the individual spam probabilities via the Fisher method, and it tells you there's a high probability that the null

hypothesis is wrong--that the message isn't a random collection of words--then it means it's likely the message is a ham. The number returned is, if not literally the probability that the message is a ham, at least a good measure of its "hamminess." Conversely, the Fisher combination of the individual ham probabilities gives you a measure of the message's "spamminess."

To get a final score, you need to combine those two measures into a single number that gives you a combined hamminess-spamminess score ranging from 0 to 1. The method recommended by Robinson is to add half the difference between the hamminess and spamminess scores to 1/2, in other words, to average the spamminess and 1 minus the hamminess. This has the nice effect that when the two scores agree (high spamminess and low hamminess, or vice versa) you'll end up with a strong indicator near either 0 or 1. But when the spamminess and hamminess scores are both high or both low, then you'll end up with a final value near 1/2, which you can treat as an "uncertain" classification.

The `score` function that implements this scheme looks like this:

```
(defun score (features)
  (let ((spam-probs ()) (ham-probs ()) (number-of-probs 0))
    (dolist (feature features)
      (unless (untrained-p feature)
        (let ((spam-prob (float (bayesian-spam-probability feature) 0.0d0)))
          (push spam-prob spam-probs)
          (push (- 1.0d0 spam-prob) ham-probs)
          (incf number-of-probs))))
    (let ((h (- 1 (fisher spam-probs number-of-probs)))
          (s (- 1 (fisher ham-probs number-of-probs))))
      (/ (+ (- 1 h) s) 2.0d0))))
```

You take a list of features and loop over them, building up two lists of probabilities, one listing the probabilities that a message containing each feature is a spam and the other that a message containing

each feature is a ham. As an optimization, you can also count the number of probabilities while looping over them and pass the count to `fisher` to avoid having to count them again in `fisher` itself. The value returned by `fisher` will be low if the individual probabilities contained too many low probabilities to have come from random text. Thus, a low `fisher` score for the spam probabilities means there were many hammy features; subtracting that score from 1 gives you a probability that the message is a ham. Conversely, subtracting the `fisher` score for the ham probabilities gives you the probability that the message was a spam. Combining those two probabilities gives you an overall spamminess score between 0 and 1.

Within the loop, you can use the function `untrained-p` to skip features extracted from the message that were never seen during training. These features will have spam counts and ham counts of zero. The `untrained-p` function is trivial.

```
(defun untrained-p (feature)
  (with-slots (spam-count ham-count) feature
    (and (zerop spam-count) (zerop ham-count))))
```

The only other new function is `fisher` itself. Assuming you already had an `inverse-chi-square` function, `fisher` is conceptually simple.

```
(defun fisher (probs number-of-probs)
  "The Fisher computation described by Robinson."
  (inverse-chi-square
   (* -2 (log (reduce #'* probs)))
   (* 2 number-of-probs)))
```

Unfortunately, there's a small problem with this straightforward implementation. While using **REDUCE** is a concise and idiomatic way of multiplying a list of numbers, in this particular application there's a danger the product will be too small a number to be represented as a

floating-point number. In that case, the result will *underflow* to zero. And if the product of the probabilities underflows, all bets are off because taking the **LOG** of zero will either signal an error or, in some implementation, result in a special negative-infinity value, which will render all subsequent calculations essentially meaningless. This is particularly unfortunate in this function because the Fisher method is most sensitive when the input probabilities are low--near zero--and therefore in the most danger of causing the multiplication to underflow.

Luckily, you can use a bit of high-school math to avoid this problem. Recall that the log of a product is the same as the sum of the logs of the factors. So instead of multiplying all the probabilities and then taking the log, you can sum the logs of each probability. And since **REDUCE** takes a `:key` keyword parameter, you can use it to perform the whole calculation. Instead of this:

```
(log (reduce #'* probs))
```

write this:

```
(reduce #'+ probs :key #'log)
```

Inverse Chi Square

The implementation of `inverse-chi-square` in this section is a fairly straightforward translation of a version written in Python by Robinson. The exact mathematical meaning of this function is beyond the scope of this book, but you can get an intuitive sense of what it does by thinking about how the values you pass to `fisher` will affect the result: the more low probabilities you pass to `fisher`, the smaller the product of the probabilities will be. The log of a small product will be a negative number with a large absolute value, which is then multiplied by -2, making it an even larger positive number. Thus, the more low probabilities were passed to `fisher`, the larger the value it'll pass to `inverse-chi-square`. Of course, the number of probabilities involved also affects the value passed to `inverse-chi-square`. Since probabilities are, by definition, less than or equal to 1, the more probabilities that go into a product, the smaller it'll be and the larger the value passed to `inverse-chi-square`. Thus, `inverse-chi-square` should return a low probability when the Fisher combined value is abnormally large for the number of probabilities that went into it. The following function does exactly that:

```
(defun inverse-chi-square (value degrees-of-freedom)
  (assert (evenp degrees-of-freedom))
  (min
   (loop with m = (/ value 2)
      for i below (/ degrees-of-freedom 2)
      for prob = (exp (- m)) then (* prob (/ m i))
      summing prob)
   1.0))
```

Recall from Chapter 10 that **`EXP`** raises *e* to the argument given. Thus, the larger `value` is, the smaller the initial value of `prob` will be.

But that initial value will then be adjusted upward slightly for each degree of freedom as long as `m` is greater than the number of degrees of freedom. Since the value returned by `inverse-chi-square` is supposed to be another probability, it's important to clamp the value returned with **`MIN`** since rounding errors in the multiplication and exponentiation may cause the **`LOOP`** to return a sum just a shade over 1.

Training the Filter

Since you wrote `classify` and `train` to take a string argument, you can test them easily at the REPL. If you haven't yet, you should switch to the package in which you've been writing this code by evaluating an **`IN-PACKAGE`** form at the REPL or using the SLIME shortcut `change-package`. To use the SLIME shortcut, type a comma at the REPL and then type the name at the prompt. Pressing Tab while typing the package name will autocomplete based on the packages your Lisp knows about. Now you can invoke any of the functions that are part of the spam application. You should first make sure the database is empty.

```
SPAM> (clear-database)
```

Now you can train the filter with some text.

```
SPAM> (train "Make money fast" 'spam)
```

And then see what the classifier thinks.

```
SPAM> (classify "Make money fast")
SPAM
SPAM> (classify "Want to go to the movies?")
UNSURE
```

While ultimately all you care about is the classification, it'd be nice to be able to see the raw score too. The easiest way to get both values without disturbing any other code is to change `classification` to return multiple values.

```
(defun classification (score)
  (values
   (cond
     ((<= score *max-ham-score*) 'ham)
     ((>= score *min-spam-score*) 'spam)
```

```
      (t 'unsure))
    score))
```

You can make this change and then recompile just this one function. Because `classify` returns whatever `classification` returns, it'll also now return two values. But since the primary return value is the same, callers of either function who expect only one value won't be affected. Now when you test `classify`, you can see exactly what score went into the classification.

```
SPAM> (classify "Make money fast")
SPAM
0.863677101854273D0
SPAM> (classify "Want to go to the movies?")
UNSURE
0.5D0
```

And now you can see what happens if you train the filter with some more ham text.

```
SPAM> (train "Do you have any money for the movies?" 'ham)
1
SPAM> (classify "Make money fast")
SPAM
0.768535121985762 6D0
```

It's still spam but a bit less certain since *money* was seen in ham text.

```
SPAM> (classify "Want to go to the movies?")
HAM
0.17482223132078922D0
```

And now this is clearly recognizable ham thanks to the presence of the word *movies*, now a hammy feature.

However, you don't really want to train the filter by hand. What you'd really like is an easy way to point it at a bunch of files and train it on them. And if you want to test how well the filter actually works, you'd like to then use it to classify another set of files of known types and

see how it does. So the last bit of code you'll write in this chapter will be a test harness that tests the filter on a corpus of messages of known types, using a certain fraction for training and then measuring how accurate the filter is when classifying the remainder.

Testing the Filter

To test the filter, you need a corpus of messages of known types. You can use messages lying around in your inbox, or you can grab one of the corpora available on the Web. For instance, the SpamAssassin corpus[12] contains several thousand messages hand classified as spam, easy ham, and hard ham. To make it easy to use whatever files you have, you can define a test rig that's driven off an array of file/type pairs. You can define a function that takes a filename and a type and adds it to the corpus like this:

```
(defun add-file-to-corpus (filename type corpus)
  (vector-push-extend (list filename type) corpus))
```

The value of `corpus` should be an adjustable vector with a fill pointer. For instance, you can make a new corpus like this:

```
(defparameter *corpus* (make-array 1000 :adjustable t :fill-pointer 0))
```

If you have the hams and spams already segregated into separate directories, you might want to add all the files in a directory as the same type. This function, which uses the `list-directory` function from Chapter 15, will do the trick:

```
(defun add-directory-to-corpus (dir type corpus)
  (dolist (filename (list-directory dir))
    (add-file-to-corpus filename type corpus)))
```

For instance, suppose you have a directory `mail` containing two subdirectories, `spam` and `ham`, each containing messages of the indicated type; you can add all the files in those two directories to `*corpus*` like this:

```
SPAM> (add-directory-to-corpus "mail/spam/" 'spam *corpus*)
NIL
SPAM> (add-directory-to-corpus "mail/ham/" 'ham *corpus*)
NIL
```

Now you need a function to test the classifier. The basic strategy will be to select a random chunk of the corpus to train on and then test the corpus by classifying the remainder of the corpus, comparing the classification returned by the `classify` function to the known classification. The main thing you want to know is how accurate the classifier is--what percentage of the messages are classified correctly? But you'll probably also be interested in what messages were misclassified and in what direction--were there more false positives or more false negatives? To make it easy to perform different analyses of the classifier's behavior, you should define the testing functions to build a list of raw results, which you can then analyze however you like.

The main testing function might look like this:

```
(defun test-classifier (corpus testing-fraction)
  (clear-database)
  (let* ((shuffled (shuffle-vector corpus))
         (size (length corpus))
         (train-on (floor (* size (- 1 testing-fraction)))))
    (train-from-corpus shuffled :start 0 :end train-on)
    (test-from-corpus shuffled :start train-on)))
```

This function starts by clearing out the feature database.[13] Then it shuffles the corpus, using a function you'll implement in a moment, and figures out, based on the `testing-fraction` parameter, how many messages it'll train on and how many it'll reserve for testing. The two helper functions `train-from-corpus` and `test-from-corpus` will both take `:start` and `:end` keyword parameters, allowing them to operate on a subsequence of the given corpus.

The `train-from-corpus` function is quite simple--simply loop over the appropriate part of the corpus, use **`DESTRUCTURING-BIND`** to extract the filename and type from the list found in each element, and then pass the text of the named file and the type to `train`. Since some mail messages, such as those with attachments, are quite large, you should limit the number of characters it'll take from the message. It'll obtain the text with a function `start-of-file`, which you'll implement in a moment, that takes a filename and a maximum number of characters to return. `train-from-corpus` looks like this:

```
(defparameter *max-chars* (* 10 1024))

(defun train-from-corpus (corpus &key (start 0) end)
  (loop for idx from start below (or end (length corpus)) do
        (destructuring-bind (file type) (aref corpus idx)
          (train (start-of-file file *max-chars*) type))))
```

The `test-from-corpus` function is similar except you want to return a list containing the results of each classification so you can analyze them after the fact. Thus, you should capture both the classification and score returned by `classify` and then collect a list of the filename, the actual type, the type returned by `classify`, and the score. To make the results more human readable, you can include keywords in the list to indicate which values are which.

```
(defun test-from-corpus (corpus &key (start 0) end)
  (loop for idx from start below (or end (length corpus)) collect
        (destructuring-bind (file type) (aref corpus idx)
          (multiple-value-bind (classification score)
              (classify (start-of-file file *max-chars*))
            (list
             :file file
             :type type
             :classification classification
             :score score)))))
```

A Couple of Utility Functions

To finish the implementation of `test-classifier`, you need to write the two utility functions that don't really have anything particularly to do with spam filtering, `shuffle-vector` and `start-of-file`.

An easy and efficient way to implement `shuffle-vector` is using the Fisher-Yates algorithm.[14] You can start by implementing a function, `nshuffle-vector`, that shuffles a vector in place. This name follows the same naming convention of other destructive functions such as **NCONC** and **NREVERSE**. It looks like this:

```
(defun nshuffle-vector (vector)
  (loop for idx downfrom (1- (length vector)) to 1
        for other = (random (1+ idx))
        do (unless (= idx other)
             (rotatef (aref vector idx) (aref vector other))))
  vector)
```

The nondestructive version simply makes a copy of the original vector and passes it to the destructive version.

```
(defun shuffle-vector (vector)
  (nshuffle-vector (copy-seq vector)))
```

The other utility function, `start-of-file`, is almost as straightforward with just one wrinkle. The most efficient way to read the contents of a file into memory is to create an array of the appropriate size and use **READ-SEQUENCE** to fill it in. So it might seem you could make a character array that's either the size of the file or the maximum number of characters you want to read, whichever is smaller. Unfortunately, as I mentioned in Chapter 14, the function **FILE-LENGTH** isn't entirely well defined when dealing with character streams since the number of characters encoded in a file can

depend on both the character encoding used and the particular text in the file. In the worst case, the only way to get an accurate measure of the number of characters in a file is to actually read the whole file. Thus, it's ambiguous what **FILE-LENGTH** should do when passed a character stream; in most implementations, **FILE-LENGTH** always returns the number of octets in the file, which may be greater than the number of characters that can be read from the file.

However, **READ-SEQUENCE** returns the number of characters actually read. So, you can attempt to read the number of characters reported by **FILE-LENGTH** and return a substring if the actual number of characters read was smaller.

```
(defun start-of-file (file max-chars)
  (with-open-file (in file)
    (let* ((length (min (file-length in) max-chars))
           (text (make-string length))
           (read (read-sequence text in)))
      (if (< read length)
        (subseq text 0 read)
        text))))
```

Analyzing the Results

Now you're ready to write some code to analyze the results generated by `test-classifier`. Recall that `test-classifier` returns the list returned by `test-from-corpus` in which each element is a plist representing the result of classifying one file. This plist contains the name of the file, the actual type of the file, the classification, and the score returned by `classify`. The first bit of analytical code you should write is a function that returns a symbol indicating whether a given result was correct, a false positive, a false negative, a missed ham, or a missed spam. You can use **`DESTRUCTURING-BIND`** to pull out the `:type` and `:classification` elements of an individual result list (using **`&allow-other-keys`** to tell **`DESTRUCTURING-BIND`** to ignore any other key/value pairs it sees) and then use nested **`ECASE`** to translate the different pairings into a single symbol.

```
(defun result-type (result)
  (destructuring-bind (&key type classification &allow-other-keys) result
    (ecase type
      (ham
       (ecase classification
         (ham 'correct)
         (spam 'false-positive)
         (unsure 'missed-ham)))
      (spam
       (ecase classification
         (ham 'false-negative)
         (spam 'correct)
         (unsure 'missed-spam))))))
```

You can test out this function at the REPL.

```
SPAM> (result-type '(:FILE #p"foo" :type ham :classification ham :score 0))
CORRECT
SPAM> (result-type '(:FILE #p"foo" :type spam :classification spam :score 0))
CORRECT
SPAM> (result-type '(:FILE #p"foo" :type ham :classification spam :score 0))
FALSE-POSITIVE
SPAM> (result-type '(:FILE #p"foo" :type spam :classification ham :score 0))
FALSE-NEGATIVE
SPAM> (result-type '(:FILE #p"foo" :type ham :classification unsure :score 0))
```

```
MISSED-HAM
SPAM> (result-type '(:FILE #p"foo" :type spam :classification unsure :score 0))
MISSED-SPAM
```

Having this function makes it easy to slice and dice the results of `test-classifier` in a variety of ways. For instance, you can start by defining predicate functions for each type of result.

```
(defun false-positive-p (result)
  (eql (result-type result) 'false-positive))

(defun false-negative-p (result)
  (eql (result-type result) 'false-negative))

(defun missed-ham-p (result)
  (eql (result-type result) 'missed-ham))

(defun missed-spam-p (result)
  (eql (result-type result) 'missed-spam))

(defun correct-p (result)
  (eql (result-type result) 'correct))
```

With those functions, you can easily use the list and sequence manipulation functions I discussed in Chapter 11 to extract and count particular kinds of results.

```
SPAM> (count-if #'false-positive-p *results*)
6
SPAM> (remove-if-not #'false-positive-p *results*)
((:FILE #p"ham/5349" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9999983107355541d0)
 (:FILE #p"ham/2746" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.6286468956619795d0)
 (:FILE #p"ham/3427" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9833753501352983d0)
 (:FILE #p"ham/7785" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9542788587998488d0)
 (:FILE #p"ham/1728" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.684339162891261d0)
 (:FILE #p"ham/10581" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9999924537959615d0))
```

You can also use the symbols returned by `result-type` as keys into a hash table or an alist. For instance, you can write a function to print a summary of the counts and percentages of each type of result using an alist that maps each type plus the extra symbol `total` to a count.

```
(defun analyze-results (results)
  (let* ((keys '(total correct false-positive
                 false-negative missed-ham missed-spam))
         (counts (loop for x in keys collect (cons x 0))))
    (dolist (item results)
```

```
      (incf (cdr (assoc 'total counts)))
      (incf (cdr (assoc (result-type item) counts))))
    (loop with total = (cdr (assoc 'total counts))
          for (label . count) in counts
          do (format t "~&~@(~a~):~20t~5d~,5t: ~6,2f%~%"
                     label count (* 100 (/ count total))))))
```

This function will give output like this when passed a list of results generated by `test-classifier`:

```
SPAM> (analyze-results *results*)
Total:               3761 : 100.00%
Correct:             3689 :  98.09%
False-positive:         4 :   0.11%
False-negative:         9 :   0.24%
Missed-ham:            19 :   0.51%
Missed-spam:           40 :   1.06%
NIL
```

And as a last bit of analysis you might want to look at why an individual message was classified the way it was. The following functions will show you:

```
(defun explain-classification (file)
  (let* ((text (start-of-file file *max-chars*))
         (features (extract-features text))
         (score (score features))
         (classification (classification score)))
    (show-summary file text classification score)
    (dolist (feature (sorted-interesting features))
      (show-feature feature))))

(defun show-summary (file text classification score)
  (format t "~&~a" file)
  (format t "~2%~a~2%" text)
  (format t "Classified as ~a with score of ~,5f~%" classification score))

(defun show-feature (feature)
  (with-slots (word ham-count spam-count) feature
    (format
     t "~&~2t~a~30thams: ~5d; spams: ~5d;~,10tprob: ~,f~%"
     word ham-count spam-count (bayesian-spam-probability feature))))

(defun sorted-interesting (features)
  (sort (remove-if #'untrained-p features) #'< :key #'bayesian-spam-probability))
```

What's Next

Obviously, you could do a lot more with this code. To turn it into a real spam-filtering application, you'd need to find a way to integrate it into your normal e-mail infrastructure. One approach that would make it easy to integrate with almost any e-mail client is to write a bit of code to act as a POP3 proxy--that's the protocol most e-mail clients use to fetch mail from mail servers. Such a proxy would fetch mail from your real POP3 server and serve it to your mail client after either tagging spam with a header that your e-mail client's filters can easily recognize or simply putting it aside. Of course, you'd also need a way to communicate with the filter about misclassifications--as long as you're setting it up as a server, you could also provide a Web interface. I'll talk about how to write Web interfaces in Chapter 26, and you'll build one, for a different application, in Chapter 29.

Or you might want to work on improving the basic classification--a likely place to start is to make `extract-features` more sophisticated. In particular, you could make the tokenizer smarter about the internal structure of e-mail--you could extract different kinds of features for words appearing in the body versus the message headers. And you could decode various kinds of message encoding such as base 64 and quoted printable since spammers often try to obfuscate their message with those encodings.

But I'll leave those improvements to you. Now you're ready to head down the path of building a streaming MP3 server, starting by writing a general-purpose library for parsing binary files.

---

[1]Available at `http://www.paulgraham.com/spam.html` and also in *Hackers & Painters: Big Ideas from the Computer Age* (O'Reilly, 2004)

[2]There has since been some disagreement over whether the technique Graham described was actually "Bayesian." However, the name has stuck and is well on its way to becoming a synonym for "statistical" when talking about spam filters.

[3]It would, however, be poor form to distribute a version of this application using a package starting with `com.gigamonkeys` since you don't control that domain.

[4]A version of CL-PPCRE is included with the book's source code available from the book's Web site. Or you can download it from Weitz's site at `http://www.weitz.de/cl-ppcre/`.

[5]The main reason to use **`PRINT-UNREADABLE-OBJECT`** is that it takes care of signaling the appropriate error if someone tries to print your object readably, such as with the `~S` **`FORMAT`** directive.

[6]**`PRINT-UNREADABLE-OBJECT`** also signals an error if it's used when the printer control variable **`*PRINT-READABLY*`** is true. Thus, a **`PRINT-OBJECT`** method consisting solely of a **`PRINT-UNREADABLE-OBJECT`** form will correctly implement the **`PRINT-OBJECT`** contract with regard to **`*PRINT-READABLY*`**.

[7]If you decide later that you do need to have different versions of `increment-feature` for different classes, you can redefine `increment-count` as a generic function and this function as a method specialized on `word-feature`.

[8]Technically, the key in each clause of a **`CASE`** or **`ECASE`** is interpreted as a *list designator*, an object that designates a list of objects. A single nonlist object, treated as a list designator, designates a list containing just that one object, while a list designates itself. Thus, each clause can have multiple keys; **`CASE`** and **`ECASE`** will select the clause whose list of keys contains the value of the key form. For example, if you wanted to make `good` a synonym for `ham` and `bad` a synonym for `spam`, you could write `increment-count` like this:

```
(defun increment-count (feature type)
  (ecase type
    ((ham good) (incf (ham-count feature)))
    ((spam bad) (incf (spam-count feature)))))
```

[9]Speaking of mathematical nuances, hard-core statisticians may be offended by the sometimes loose use of the word *probability* in this chapter. However, since even the pros, who are divided between the Bayesians and the frequentists, can't agree on what a

probability is, I'm not going to worry about it. This is a book about programming, not statistics.

[10]Robinson's articles that directly informed this chapter are "A Statistical Approach to the Spam Problem" (published in the *Linux Journal* and available at `http://www.linuxjournal.com/ article.php?sid=6467` and in a shorter form on Robinson's blog at `http://radio.weblogs.com/ 0101454/stories/2002/09/16/spamDetection.html`) and "Why Chi? Motivations for the Use of Fisher's Inverse Chi-Square Procedure in Spam Classification" (available at `http://garyrob.blogs.com/ whychi93.pdf`). Another article that may be useful is "Handling Redundancy in Email Token Probabilities" (available at `http://garyrob.blogs.com//handlingtokenredundancy94.pdf`). The archived mailing lists of the SpamBayes project (`http://spambayes.sourceforge.net/`) also contain a lot of useful information about different algorithms and approaches to testing spam filters.

[11]Techniques that combine nonindependent probabilities as though they were, in fact, independent, are called *naive Bayesian*. Graham's original proposal was essentially a naive Bayesian classifier with some "empirically derived" constant factors thrown in.

[12]Several spam corpora including the SpamAssassin corpus are linked to from `http://nexp.cs.pdx.edu/~psam/cgi-bin/view/PSAM/CorpusSets`.

[13]If you wanted to conduct a test without disturbing the existing database, you could bind `*feature-database*`, `*total-spams*`, and `*total-hams*` with a **`LET`**, but then you'd have no way of looking at the database after the fact--unless you returned the values you used within the function.

[14]This algorithm is named for the same Fisher who invented the method used for combining probabilities and for Frank Yates, his coauthor of the book *Statistical Tables for Biological, Agricultural and Medical Research* (Oliver & Boyd, 1938) in which, according to Knuth, they provided the first published description of the algorithm.

Copyright © 2003-2005, Peter Seibel

24. Practical: Parsing Binary Files

In this chapter I'll show you how to build a library that you can use to write code for reading and writing binary files. You'll use this library in Chapter 25 to write a parser for ID3 tags, the mechanism used to store metadata such as artist and album names in MP3 files. This library is also an example of how to use macros to extend the language with new constructs, turning it into a special-purpose language for solving a particular problem, in this case reading and writing binary data. Because you'll develop the library a bit at a time, including several partial versions, it may seem you're writing a lot of code. But when all is said and done, the whole library is fewer than 150 lines of code, and the longest macro is only 20 lines long.

Binary Files

At a sufficiently low level of abstraction, all files are "binary" in the sense that they just contain a bunch of numbers encoded in binary form. However, it's customary to distinguish between *text files*, where all the numbers can be interpreted as characters representing human-readable text, and *binary files*, which contain data that, if interpreted as characters, yields nonprintable characters.[1]

Binary file formats are usually designed to be both compact and efficient to parse--that's their main advantage over text-based formats. To meet both those criteria, they're usually composed of on-disk structures that are easily mapped to data structures that a program might use to represent the same data in memory.[2]

The library will give you an easy way to define the mapping between the on-disk structures defined by a binary file format and in-memory Lisp objects. Using the library, it should be easy to write a program that can read a binary file, translating it into Lisp objects that you can manipulate, and then write back out to another properly formatted binary file.

Binary Format Basics

The starting point for reading and writing binary files is to open the file for reading or writing individual bytes. As I discussed in Chapter 14, both `OPEN` and `WITH-OPEN-FILE` accept a keyword argument, `:element-type`, that controls the basic unit of transfer for the stream. When you're dealing with binary files, you'll specify `(unsigned-byte 8)`. An input stream opened with such an `:element-type` will return an integer between 0 and 255 each time it's passed to `READ-BYTE`. Conversely, you can write bytes to an `(unsigned-byte 8)` output stream by passing numbers between 0 and 255 to `WRITE-BYTE`.

Above the level of individual bytes, most binary formats use a smallish number of primitive data types--numbers encoded in various ways, textual strings, bit fields, and so on--which are then composed into more complex structures. So your first task is to define a framework for writing code to read and write the primitive data types used by a given binary format.

To take a simple example, suppose you're dealing with a binary format that uses an unsigned 16-bit integer as a primitive data type. To read such an integer, you need to read the two bytes and then combine them into a single number by multiplying one byte by 256, a.k.a. 2^8, and adding it to the other byte. For instance, assuming the binary format specifies that such 16-bit quantities are stored in *big-endian*[3] form, with the most significant byte first, you can read such a number with this function:

```
(defun read-u2 (in)
  (+ (* (read-byte in) 256) (read-byte in)))
```

However, Common Lisp provides a more convenient way to perform this kind of bit twiddling. The function **LDB**, whose name stands for load byte, can be used to extract and set (with **SETF**) any number of contiguous bits from an integer.[4] The number of bits and their position within the integer is specified with a *byte specifier* created with the **BYTE** function. **BYTE** takes two arguments, the number of bits to extract (or set) and the position of the rightmost bit where the least significant bit is at position zero. **LDB** takes a byte specifier and the integer from which to extract the bits and returns the positive integer represented by the extracted bits. Thus, you can extract the least significant octet of an integer like this:

```
(ldb (byte 8 0) #xabcd) ==> 205 ; 205 is #xcd
```

To get the next octet, you'd use a byte specifier of `(byte 8 8)` like this:

```
(ldb (byte 8 8) #xabcd) ==> 171 ; 171 is #xab
```

You can use **LDB** with **SETF** to set the specified bits of an integer stored in a **SETF**able place.

```
CL-USER> (defvar *num* 0)
*NUM*
CL-USER> (setf (ldb (byte 8 0) *num*) 128)
128
CL-USER> *num*
128
CL-USER> (setf (ldb (byte 8 8) *num*) 255)
255
CL-USER> *num*
65408
```

Thus, you can also write `read-u2` like this:[5]

```
(defun read-u2 (in)
  (let ((u2 0))
```

```
    (setf (ldb (byte 8 8) u2) (read-byte in))
    (setf (ldb (byte 8 0) u2) (read-byte in))
    u2))
```

To write a number out as a 16-bit integer, you need to extract the individual 8-bit bytes and write them one at a time. To extract the individual bytes, you just need to use **LDB** with the same byte specifiers.

```
(defun write-u2 (out value)
  (write-byte (ldb (byte 8 8) value) out)
  (write-byte (ldb (byte 8 0) value) out))
```

Of course, you can also encode integers in many other ways--with different numbers of bytes, with different endianness, and in signed and unsigned format.

Strings in Binary Files

Textual strings are another kind of primitive data type you'll find in many binary formats. When you read files one byte at a time, you can't read and write strings directly--you need to decode and encode them one byte at a time, just as you do with binary-encoded numbers. And just as you can encode an integer in several ways, you can encode a string in many ways. To start with, the binary format must specify how individual characters are encoded.

To translate bytes to characters, you need to know both what character *code* and what character *encoding* you're using. A character code defines a mapping from positive integers to characters. Each number in the mapping is called a *code point*. For instance, ASCII is a character code that maps the numbers from 0-127 to particular characters used in the Latin alphabet. A character encoding, on the other hand, defines how the code points are represented as a sequence of bytes in a byte-oriented medium such as a file. For codes that use eight or fewer bits, such as ASCII and ISO-8859-1, the encoding is trivial--each numeric value is encoded as a single byte.

Nearly as straightforward are pure double-byte encodings, such as UCS-2, which map between 16-bit values and characters. The only reason double-byte encodings can be more complex than single-byte encodings is that you may also need to know whether the 16-bit values are supposed to be encoded in big-endian or little-endian format.

Variable-width encodings use different numbers of octets for different numeric values, making them more complex but allowing them to be more compact in many cases. For instance, UTF-8, an encoding designed for use with the Unicode character code, uses a single octet to encode the values 0-127 while using up to four octets to encode values up to 1,114,111.[6]

Since the code points from 0-127 map to the same characters in Unicode as they do in ASCII, a UTF-8 encoding of text consisting only of characters also in ASCII is the same as the ASCII encoding. On the other hand, texts consisting mostly of characters requiring four bytes in UTF-8 could be more compactly encoded in a straight double-byte encoding.

Common Lisp provides two functions for translating between numeric character codes and character objects: `CODE-CHAR`, which takes an numeric code and returns as a character, and `CHAR-CODE`, which takes a character and returns its numeric code. The language standard doesn't specify what character encoding an implementation must use, so there's no guarantee you can represent every character that can possibly be encoded in a given file format as a Lisp character. However, almost all contemporary Common Lisp implementations use ASCII, ISO-8859-1, or Unicode as their native character code. Because Unicode is a superset ofISO-8859-1, which is in turn a superset of ASCII, if you're using a Unicode Lisp, `CODE-CHAR` and `CHAR-CODE` can be used directly for translating any of those three character codes.[7]

In addition to specifying a character encoding, a string encoding must also specify how to encode the length of the string. Three techniques are typically used in binary file formats.

The simplest is to not encode it but to let it be implicit in the position of the string in some larger structure: a particular element of a file may always be a string of a certain length, or a string may be the last element of a variable-length data structure whose overall size determines how many bytes are left to read as string data. Both these techniques are used in ID3 tags, as you'll see in the next chapter.

The other two techniques can be used to encode variable-length strings without relying on context. One is to encode the length of the string followed by the character data--the parser reads an integer value (in some specified integer format) and then reads that number of characters. Another is to write the character data followed by a delimiter that can't appear in the string such as a null character.

The different representations have different advantages and disadvantages, but when you're dealing with already specified binary formats, you won't have any control over which encoding is used. However, none of the encodings is particularly more difficult to read and write than any other. Here, as an example, is a function that reads a null-terminated ASCII string, assuming your Lisp implementation uses ASCII or one of its supersets such as ISO-8859-1 or full Unicode as its native character encoding:

```
(defconstant +null+ (code-char 0))

(defun read-null-terminated-ascii (in)
  (with-output-to-string (s)
```

```
    (loop for char = (code-char (read-byte in))
          until (char= char +null+) do (write-char char s))))
```

The **WITH-OUTPUT-TO-STRING** macro, which I mentioned in Chapter 14, is an easy way to build up a string when you don't know how long it'll be. It creates a **STRING-STREAM** and binds it to the variable name specified, `S` in this case. All characters written to the stream are collected into a string, which is then returned as the value of the **WITH-OUTPUT-TO-STRING** form.

To write a string back out, you just need to translate the characters back to numeric values that can be written with **WRITE-BYTE** and then write the null terminator after the string contents.

```
(defun write-null-terminated-ascii (string out)
  (loop for char across string
        do (write-byte (char-code char) out))
  (write-byte (char-code +null+) out))
```

As these examples show, the main intellectual challenge--such as it is--of reading and writing primitive elements of binary files is understanding how exactly to interpret the bytes that appear in a file and to map them to Lisp data types. If a binary file format is well specified, this should be a straightforward proposition. Actually writing functions to read and write a particular encoding is, as they say, a simple matter of programming.

Now you can turn to the issue of reading and writing more complex on-disk structures and how to map them to Lisp objects.

Composite Structures

Since binary formats are usually used to represent data in a way that makes it easy to map to in-memory data structures, it should come as no surprise that composite on-disk structures are usually defined in ways similar to the way programming languages define in-memory structures. Usually a composite on-disk structure will consist of a number of named parts, each of which is itself either a primitive type such as a number or a string, another composite structure, or possibly a collection of such values.

For instance, an ID3 tag defined in the 2.2 version of the specification consists of a header made up of a three-character ISO-8859-1 string, which is always "ID3"; two one-byte unsigned integers that specify the major version and revision of the specification; eight bits worth of boolean flags; and four bytes that encode the size of the tag in an encoding particular to the ID3 specification. Following the header is a list of *frames*, each of which has its own internal structure. After the frames are as many null bytes as are necessary to pad the tag out to the size specified in the header.

If you look at the world through the lens of object orientation, composite structures look a lot like classes. For instance, you could write a class to represent an ID3 tag.

```
(defclass id3-tag ()
  ((identifier    :initarg :identifier    :accessor identifier)
   (major-version :initarg :major-version :accessor major-version)
   (revision      :initarg :revision      :accessor revision)
   (flags         :initarg :flags         :accessor flags)
   (size          :initarg :size          :accessor size)
   (frames        :initarg :frames        :accessor frames)))
```

An instance of this class would make a perfect repository to hold the data needed to represent an ID3 tag. You could then write functions to read and write instances of this class. For example, assuming the existence of certain other functions for reading the appropriate primitive data types, a `read-id3-tag` function might look like this:

```
(defun read-id3-tag (in)
  (let ((tag (make-instance 'id3-tag)))
    (with-slots (identifier major-version revision flags size frames) tag
      (setf identifier    (read-iso-8859-1-string in :length 3))
      (setf major-version (read-u1 in))
      (setf revision      (read-u1 in))
      (setf flags         (read-u1 in))
      (setf size          (read-id3-encoded-size in))
      (setf frames        (read-id3-frames in :tag-size size)))
    tag))
```

The `write-id3-tag` function would be structured similarly--you'd use the appropriate `write-*` functions to write out the values stored in the slots of the `id3-tag` object.

It's not hard to see how you could write the appropriate classes to represent all the composite data structures in a specification along with `read-foo` and `write-foo` functions for each class and for necessary primitive types. But it's also easy to tell that all the reading and writing functions are going to be pretty similar, differing only in the specifics of what types they read and the names of the slots they store them in. It's particularly irksome when you consider that in the ID3 specification it takes about four lines of text to specify the structure of an ID3 tag, while you've already written eighteen lines of code and haven't even written `write-id3-tag` yet.

What you'd really like is a way to describe the structure of something like an ID3 tag in a form that's as compressed as the specification's pseudocode yet that can also be expanded into code that defines the

`id3-tag` class *and* the functions that translate between bytes on disk and instances of the class. Sounds like a job for a macro.

Designing the Macros

Since you already have a rough idea what code your macros will need to generate, the next step, according to the process for writing a macro I outlined in Chapter 8, is to switch perspectives and think about what a call to the macro should look like. Since the goal is to be able to write something as compressed as the pseudocode in the ID3 specification, you can start there. The header of an ID3 tag is specified like this:

```
ID3/file identifier      "ID3"
ID3 version              $02 00
ID3 flags                %xx000000
ID3 size             4 * %0xxxxxxx
```

In the notation of the specification, this means the "file identifier" slot of an ID3 tag is the string "ID3" in ISO-8859-1 encoding. The version consists of two bytes, the first of which--for this version of the specification--has the value 2 and the second of which--again for this version of the specification--is 0. The flags slot is eight bits, of which all but the first two are 0, and the size consists of four bytes, each of which has a 0 in the most significant bit.

Some information isn't captured by this pseudocode. For instance, exactly how the four bytes that encode the size are to be interpreted is described in a few lines of prose. Likewise, the spec describes in prose how the frame and subsequent padding is stored after this header. But most of what you need to know to be able to write code to read and write an ID3 tag is specified by this pseudocode. Thus, you ought to be able to write an s-expression version of this pseudocode and have it expanded into the class and function

definitions you'd otherwise have to write by hand--something, perhaps, like this:

```
(define-binary-class id3-tag
  ((file-identifier (iso-8859-1-string :length 3))
   (major-version   u1)
   (revision        u1)
   (flags           u1)
   (size            id3-tag-size)
   (frames          (id3-frames :tag-size size))))
```

The basic idea is that this form defines a class `id3-tag` similar to the way you could with **DEFCLASS**, but instead of specifying things such as `:initarg` and `:accessors`, each slot specification consists of the name of the slot--`file-identifier`, `major-version`, and so on--and information about how that slot is represented on disk. Since this is just a bit of fantasizing, you don't have to worry about exactly how the macro `define-binary-class` will know what to do with expressions such as `(iso-8859-1-string :length 3)`, `u1`, `id3-tag-size`, and `(id3-frames :tag-size size)`; as long as each expression contains the information necessary to know how to read and write a particular data encoding, you should be okay.

Making the Dream a Reality

Okay, enough fantasizing about good-looking code; now you need to get to work writing `define-binary-class`--writing the code that will turn that concise expression of what an ID3 tag looks like into code that can represent one in memory, read one off disk, and write it back out.

To start with, you should define a package for this library. Here's the package file that comes with the version you can download from the book's Web site:

```
(in-package :cl-user)

(defpackage :com.gigamonkeys.binary-data
  (:use :common-lisp :com.gigamonkeys.macro-utilities)
  (:export :define-binary-class
           :define-tagged-binary-class
           :define-binary-type
           :read-value
           :write-value
           :*in-progress-objects*
           :parent-of-type
           :current-binary-object
           :+null+))
```

The `COM.GIGAMONKEYS.MACRO-UTILITIES` package contains the `with-gensyms` and `once-only` macros from Chapter 8.

Since you already have a handwritten version of the code you want to generate, it shouldn't be too hard to write such a macro. Just take it in small pieces, starting with a version of `define-binary-class` that generates just the **`DEFCLASS`** form.

If you look back at the `define-binary-class` form, you'll see that it takes two arguments, the name `id3-tag` and a list of slot specifiers,

each of which is itself a two-item list. From those pieces you need to build the appropriate **DEFCLASS** form. Clearly, the biggest difference between the `define-binary-class` form and a proper **DEFCLASS** form is in the slot specifiers. A single slot specifier from `define-binary-class` looks something like this:

```
(major-version u1)
```

But that's not a legal slot specifier for a **DEFCLASS**. Instead, you need something like this:

```
(major-version :initarg :major-version :accessor major-version)
```

Easy enough. First define a simple function to translate a symbol to the corresponding keyword symbol.

```
(defun as-keyword (sym) (intern (string sym) :keyword))
```

Now define a function that takes a `define-binary-class` slot specifier and returns a **DEFCLASS** slot specifier.

```
(defun slot->defclass-slot (spec)
  (let ((name (first spec)))
    `(,name :initarg ,(as-keyword name) :accessor ,name)))
```

You can test this function at the REPL after switching to your new package with a call to **IN-PACKAGE**.

```
BINARY-DATA> (slot->defclass-slot '(major-version u1))
(MAJOR-VERSION :INITARG :MAJOR-VERSION :ACCESSOR MAJOR-VERSION)
```

Looks good. Now the first version of `define-binary-class` is trivial.

```
(defmacro define-binary-class (name slots)
  `(defclass ,name ()
     ,(mapcar #'slot->defclass-slot slots)))
```

This is simple template-style macro--`define-binary-class` generates a **`DEFCLASS`** form by interpolating the name of the class and a list of slot specifiers constructed by applying `slot->defclass-slot` to each element of the list of slots specifiers from the `define-binary-class` form.

To see exactly what code this macro generates, you can evaluate this expression at the REPL.

```
(macroexpand-1 '(define-binary-class id3-tag
  ((identifier      (iso-8859-1-string :length 3))
   (major-version   u1)
   (revision        u1)
   (flags           u1)
   (size            id3-tag-size)
   (frames          (id3-frames :tag-size size)))))
```

The result, slightly reformatted here for better readability, should look familiar since it's exactly the class definition you wrote by hand earlier:

```
(defclass id3-tag ()
  ((identifier      :initarg :identifier    :accessor identifier)
   (major-version   :initarg :major-version :accessor major-version)
   (revision        :initarg :revision      :accessor revision)
   (flags           :initarg :flags         :accessor flags)
   (size            :initarg :size          :accessor size)
   (frames          :initarg :frames        :accessor frames)))
```

Reading Binary Objects

Next you need to make `define-binary-class` also generate a function that can read an instance of the new class. Looking back at the `read-id3-tag` function you wrote before, this seems a bit trickier, as the `read-id3-tag` wasn't quite so regular--to read each slot's value, you had to call a different function. Not to mention, the name of the function, `read-id3-tag`, while derived from the name of the class you're defining, isn't one of the arguments to `define-binary-class` and thus isn't available to be interpolated into a template the way the class name was.

You could deal with both of those problems by devising and following a naming convention so the macro can figure out the name of the function to call based on the name of the type in the slot specifier. However, this would require `define-binary-class` to generate the name `read-id3-tag`, which is possible but a bad idea. Macros that create global definitions should generally use only names passed to them by their callers; macros that generate names under the covers can cause hard-to-predict--and hard-to-debug--name conflicts when the generated names happen to be the same as names used elsewhere.[8]

You can avoid both these inconveniences by noticing that all the functions that read a particular type of value have the same fundamental purpose, to read a value of a specific type from a stream. Speaking colloquially, you might say they're all instances of a single generic operation. And the colloquial use of the word *generic* should lead you directly to the solution to your problem: instead of

defining a bunch of independent functions, all with different names, you can define a single generic function, `read-value`, with methods specialized to read different types of values.

That is, instead of defining functions `read-iso-8859-1-string` and `read-u1`, you can define `read-value` as a generic function taking two required arguments, a type and a stream, and possibly some keyword arguments.

```
(defgeneric read-value (type stream &key)
  (:documentation "Read a value of the given type from the stream."))
```

By specifying **`&key`** without any actual keyword parameters, you allow different methods to define their own **`&key`** parameters without requiring them to do so. This does mean every method specialized on `read-value` will have to include either **`&key`** or an **`&rest`** parameter in its parameter list to be compatible with the generic function.

Then you'll define methods that use **`EQL`** specializers to specialize the type argument on the name of the type you want to read.

```
(defmethod read-value ((type (eql 'iso-8859-1-string)) in &key length) ...)

(defmethod read-value ((type (eql 'u1)) in &key) ...)
```

Then you can make `define-binary-class` generate a `read-value` method specialized on the type name `id3-tag`, and that method can be implemented in terms of calls to `read-value` with the appropriate slot types as the first argument. The code you want to generate is going to look like this:

```
(defmethod read-value ((type (eql 'id3-tag)) in &key)
  (let ((object (make-instance 'id3-tag)))
    (with-slots (identifier major-version revision flags size frames) object
      (setf identifier    (read-value 'iso-8859-1-string in :length 3))
      (setf major-version (read-value 'u1 in))
      (setf revision      (read-value 'u1 in))
```

```
    (setf flags         (read-value 'u1 in))
    (setf size          (read-value 'id3-encoded-size in))
    (setf frames        (read-value 'id3-frames in :tag-size size)))
  object))
```

So, just as you needed a function to translate a `define-binary-class` slot specifier to a **`DEFCLASS`** slot specifier in order to generate the **`DEFCLASS`** form, now you need a function that takes a `define-binary-class` slot specifier and generates the appropriate **`SETF`** form, that is, something that takes this:

```
(identifier (iso-8859-1-string :length 3))
```

and returns this:

```
(setf identifier (read-value 'iso-8859-1-string in :length 3))
```

However, there's a difference between this code and the **`DEFCLASS`** slot specifier: it includes a reference to a variable `in`--the method parameter from the `read-value` method--that wasn't derived from the slot specifier. It doesn't have to be called `in`, but whatever name you use has to be the same as the one used in the method's parameter list and in the other calls to `read-value`. For now you can dodge the issue of where that name comes from by defining `slot->read-value` to take a second argument of the name of the stream variable.

```
(defun slot->read-value (spec stream)
  (destructuring-bind (name (type &rest args)) (normalize-slot-spec spec)
    `(setf ,name (read-value ',type ,stream ,@args))))
```

The function `normalize-slot-spec` normalizes the second element of the slot specifier, converting a symbol like `u1` to the list `(u1)` so the **`DESTRUCTURING-BIND`** can parse it. It looks like this:

```
(defun normalize-slot-spec (spec)
  (list (first spec) (mklist (second spec))))

(defun mklist (x) (if (listp x) x (list x)))
```

You can test `slot->read-value` with each type of slot specifier.

```
BINARY-DATA> (slot->read-value '(major-version u1) 'stream)
(SETF MAJOR-VERSION (READ-VALUE 'U1 STREAM))
BINARY-DATA> (slot->read-value '(identifier (iso-8859-1-string :length 3)) 'stream)
(SETF IDENTIFIER (READ-VALUE 'ISO-8859-1-STRING STREAM :LENGTH 3))
```

With these functions you're ready to add `read-value` to `define-binary-class`. If you take the handwritten `read-value` method and strip out anything that's tied to a particular class, you're left with this skeleton:

```
(defmethod read-value ((type (eql ...)) stream &key)
  (let ((object (make-instance ...)))
    (with-slots (...) object
      ...
    object)))
```

All you need to do is add this skeleton to the `define-binary-class` template, replacing ellipses with code that fills in the skeleton with the appropriate names and code. You'll also want to replace the variables `type`, `stream`, and `object` with gensymed names to avoid potential conflicts with slot names,[9] which you can do with the `with-gensyms` macro from Chapter 8.

Also, because a macro must expand into a single form, you need to wrap some form around the **DEFCLASS** and **DEFMETHOD**. **PROGN** is the customary form to use for macros that expand into multiple definitions because of the special treatment it gets from the file compiler when appearing at the top level of a file, as I discussed in Chapter 20.

So, you can change `define-binary-class` as follows:

```
(defmacro define-binary-class (name slots)
  (with-gensyms (typevar objectvar streamvar)
    `(progn
       (defclass ,name ()
```

```
  ,(mapcar #'slot->defclass-slot slots))

(defmethod read-value ((,typevar (eql ',name)) ,streamvar &key)
  (let ((,objectvar (make-instance ',name)))
    (with-slots ,(mapcar #'first slots) ,objectvar
      ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots))
    ,objectvar)))))
```

Writing Binary Objects

Generating code to write out an instance of a binary class will proceed similarly. First you can define a `write-value` generic function.

```
(defgeneric write-value (type stream value &key)
  (:documentation "Write a value as the given type to the stream."))
```

Then you define a helper function that translates a `define-binary-class` slot specifier into code that writes out the slot using `write-value`. As with the `slot->read-value` function, this helper function needs to take the name of the stream variable as an argument.

```
(defun slot->write-value (spec stream)
  (destructuring-bind (name (type &rest args)) (normalize-slot-spec spec)
    `(write-value ',type ,stream ,name ,@args)))
```

Now you can add a `write-value` template to the `define-binary-class` macro.

```
(defmacro define-binary-class (name slots)
  (with-gensyms (typevar objectvar streamvar)
    `(progn
       (defclass ,name ()
         ,(mapcar #'slot->defclass-slot slots))

       (defmethod read-value ((,typevar (eql ',name)) ,streamvar &key)
         (let ((,objectvar (make-instance ',name)))
           (with-slots ,(mapcar #'first slots) ,objectvar
             ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots))
           ,objectvar))

       (defmethod write-value ((,typevar (eql ',name)) ,streamvar ,objectvar &key)
         (with-slots ,(mapcar #'first slots) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```

Adding Inheritance and Tagged Structures

While this version of `define-binary-class` will handle stand-alone structures, binary file formats often define on-disk structures that would be natural to model with subclasses and superclasses. So you might want to extend `define-binary-class` to support inheritance.

A related technique used in many binary formats is to have several on-disk structures whose exact type can be determined only by reading some data that indicates how to parse the following bytes. For instance, the frames that make up the bulk of an ID3 tag all share a common header structure consisting of a string identifier and a length. To read a frame, you need to read the identifier and use its value to determine what kind of frame you're looking at and thus how to parse the body of the frame.

The current `define-binary-class` macro has no way to handle this kind of reading--you could use `define-binary-class` to define a class to represent each kind of frame, but you'd have no way to know what type of frame to read without reading at least the identifier. And if other code reads the identifier in order to determine what type to pass to `read-value`, then that will break `read-value` since it's expecting to read all the data that makes up the instance of the class it instantiates.

You can solve this problem by adding inheritance to `define-binary-class` and then writing another macro, `define-tagged-binary-class`, for defining "abstract" classes that aren't instantiated directly but that can be specialized on by

`read-value` methods that know how to read enough data to determine what kind of class to create.

The first step to adding inheritance to `define-binary-class` is to add a parameter to the macro to accept a list of superclasses.

```
(defmacro define-binary-class (name (&rest superclasses) slots) ...
```

Then, in the **`DEFCLASS`** template, interpolate that value instead of the empty list.

```
(defclass ,name ,superclasses
  ...)
```

However, there's a bit more to it than that. You also need to change the `read-value` and `write-value` methods so the methods generated when defining a superclass can be used by the methods generated as part of a subclass to read and write inherited slots.

The current way `read-value` works is particularly problematic since it instantiates the object before filling it in--obviously, you can't have the method responsible for reading the superclass's fields instantiate one object while the subclass's method instantiates and fills in a different object.

You can fix that problem by splitting `read-value` into two parts--one responsible for instantiating the correct kind of object and another responsible for filling slots in an existing object. On the writing side it's a bit simpler, but you can use the same technique.

So you'll define two new generic functions, `read-object` and `write-object`, that will both take an existing object and a stream. Methods on these generic functions will be responsible for reading or

writing the slots specific to the class of the object on which they're specialized.

```
(defgeneric read-object (object stream)
  (:method-combination progn :most-specific-last)
  (:documentation "Fill in the slots of object from stream."))

(defgeneric write-object (object stream)
  (:method-combination progn :most-specific-last)
  (:documentation "Write out the slots of object to the stream."))
```

Defining these generic functions to use the **PROGN** method combination with the option `:most-specific-last` allows you to define methods that specialize `object` on each binary class and have them deal only with the slots actually defined in that class; the **PROGN** method combination will combine all the applicable methods so the method specialized on the least specific class in the hierarchy runs first, reading or writing the slots defined in that class, then the method specialized on next least specific subclass, and so on. And since all the heavy lifting for a specific class is now going to be done by `read-object` and `write-object`, you don't even need to define specialized `read-value` and `write-value` methods; you can define default methods that assume the type argument is the name of a binary class.

```
(defmethod read-value ((type symbol) stream &key)
  (let ((object (make-instance type)))
    (read-object object stream)
    object))

(defmethod write-value ((type symbol) stream value &key)
  (assert (typep value type))
  (write-object value stream))
```

Note how you can use **MAKE-INSTANCE** as a generic object factory--while you normally call **MAKE-INSTANCE** with a quoted symbol as the first argument because you normally know exactly what class you want to instantiate, you can use any expression that evaluates to a

class name such as, in this case, the `type` parameter in the `read-value` method.

The actual changes to `define-binary-class` to define methods on `read-object` and `write-object` rather than `read-value` and `write-value` are fairly minor.

```
(defmacro define-binary-class (name superclasses slots)
  (with-gensyms (objectvar streamvar)
    `(progn
       (defclass ,name ,superclasses
         ,(mapcar #'slot->defclass-slot slots))

       (defmethod read-object progn ((,objectvar ,name) ,streamvar)
         (with-slots ,(mapcar #'first slots) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots)))

       (defmethod write-object progn ((,objectvar ,name) ,streamvar)
         (with-slots ,(mapcar #'first slots) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```

Keeping Track of Inherited Slots

This definition will work for many purposes. However, it doesn't handle one fairly common situation, namely, when you have a subclass that needs to refer to inherited slots in its own slot specifications. For instance, with the current definition of `define-binary-class`, you can define a single class like this:

```
(define-binary-class generic-frame ()
  ((id (iso-8859-1-string :length 3))
   (size u3)
   (data (raw-bytes :bytes size))))
```

The reference to `size` in the specification of `data` works the way you'd expect because the expressions that read and write the `data` slot are wrapped in a **`WITH-SLOTS`** that lists all the object's slots. However, if you try to split that class into two classes like this:

```
(define-binary-class frame ()
  ((id (iso-8859-1-string :length 3))
   (size u3)))

(define-binary-class generic-frame (frame)
  ((data (raw-bytes :bytes size))))
```

you'll get a compile-time warning when you compile the `generic-frame` definition and a runtime error when you try to use it because there will be no lexically apparent variable `size` in the `read-object` and `write-object` methods specialized on `generic-frame`.

What you need to do is keep track of the slots defined by each binary class and then include inherited slots in the **`WITH-SLOTS`** forms in the `read-object` and `write-object` methods.

The easiest way to keep track of information like this is to hang it off the symbol that names the class. As I discussed in Chapter 21, every symbol object has an associated property list, which can be accessed via the functions **SYMBOL-PLIST** and **GET**. You can associate arbitrary key/value pairs with a symbol by adding them to its property list with **SETF** of **GET**. For instance, if the binary class `foo` defines three slots--`x`, `y`, and `z`--you can keep track of that fact by adding a `slots` key to the symbol `foo`'s property list with the value `(x y z)` with this expression:

```
(setf (get 'foo 'slots) '(x y z))
```

You want this bookkeeping to happen as part of evaluating the `define-binary-class` of `foo`. However, it's not clear where to put the expression. If you evaluate it when you compute the macro's expansion, it'll get evaluated when you compile the `define-binary-class` form but not if you later load a file that contains the resulting compiled code. On the other hand, if you include the expression in the expansion, then it *won't* be evaluated during compilation, which means if you compile a file with several `define-binary-class` forms, none of the information about what classes define what slots will be available until the whole file is loaded, which is too late.

This is what the special operator **EVAL-WHEN** I discussed in Chapter 20 is for. By wrapping a form in an **EVAL-WHEN**, you can control whether it's evaluated at compile time, when the compiled code is loaded, or both. For cases like this where you want to squirrel away some information during the compilation of a macro form that you also

want to be available after the compiled form is loaded, you should wrap it in an **EVAL-WHEN** like this:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf (get 'foo 'slots) '(x y z)))
```

and include the **EVAL-WHEN** in the expansion generated by the macro. Thus, you can save both the slots and the direct superclasses of a binary class by adding this form to the expansion generated by `define-binary-class`:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf (get ',name 'slots) ',(mapcar #'first slots))
  (setf (get ',name 'superclasses) ',superclasses))
```

Now you can define three helper functions for accessing this information. The first simply returns the slots directly defined by a binary class. It's a good idea to return a copy of the list since you don't want other code to modify the list of slots after the binary class has been defined.

```
(defun direct-slots (name)
  (copy-list (get name 'slots)))
```

The next function returns the slots inherited from other binary classes.

```
(defun inherited-slots (name)
  (loop for super in (get name 'superclasses)
        nconc (direct-slots super)
        nconc (inherited-slots super)))
```

Finally, you can define a function that returns a list containing the names of all directly defined and inherited slots.

```
(defun all-slots (name)
  (nconc (direct-slots name) (inherited-slots name)))
```

When you're computing the expansion of a `define-generic-binary-class` form, you want to generate a **`WITH-SLOTS`** form that contains the names of all the slots defined in the new class and all its superclasses. However, you can't use `all-slots` while you're generating the expansion since the information won't be available until after the expansion is compiled. Instead, you should use the following function, which takes the list of slot specifiers and superclasses passed to `define-generic-binary-class` and uses them to compute the list of all the new class's slots:

```
(defun new-class-all-slots (slots superclasses)
  (nconc (mapcan #'all-slots superclasses) (mapcar #'first slots)))
```

With these functions defined, you can change `define-binary-class` to store the information about the class currently being defined and to use the already stored information about the superclasses' slots to generate the **`WITH-SLOTS`** forms you want like this:

```
(defmacro define-binary-class (name (&rest superclasses) slots)
  (with-gensyms (objectvar streamvar)
    `(progn
       (eval-when (:compile-toplevel :load-toplevel :execute)
         (setf (get ',name 'slots) ',(mapcar #'first slots))
         (setf (get ',name 'superclasses) ',superclasses))

       (defclass ,name ,superclasses
         ,(mapcar #'slot->defclass-slot slots))

       (defmethod read-object progn ((,objectvar ,name) ,streamvar)
         (with-slots ,(new-class-all-slots slots superclasses) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots)))

       (defmethod write-object progn ((,objectvar ,name) ,streamvar)
         (with-slots ,(new-class-all-slots slots superclasses) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```

Tagged Structures

With the ability to define binary classes that extend other binary classes, you're ready to define a new macro for defining classes to represent "tagged" structures. The strategy for reading tagged structures will be to define a specialized `read-value` method that knows how to read the values that make up the start of the structure and then use those values to determine what subclass to instantiate. It'll then make an instance of that class with **`MAKE-INSTANCE`**, passing the already read values as initargs, and pass the object to `read-object`, allowing the actual class of the object to determine how the rest of the structure is read.

The new macro, `define-tagged-binary-class`, will look like `define-binary-class` with the addition of a `:dispatch` option used to specify a form that should evaluate to the name of a binary class. The `:dispatch` form will be evaluated in a context where the names of the slots defined by the tagged class are bound to variables that hold the values read from the file. The class whose name it returns must accept initargs corresponding to the slot names defined by the tagged class. This is easily ensured if the `:dispatch` form always evaluates to the name of a class that subclasses the tagged class.

For instance, supposing you have a function, `find-frame-class`, that will map a string identifier to a binary class representing a particular kind of ID3 frame, you might define a tagged binary class, `id3-frame`, like this:

```
(define-tagged-binary-class id3-frame ()
  ((id   (iso-8859-1-string :length 3))
   (size u3))
  (:dispatch (find-frame-class id)))
```

The expansion of a `define-tagged-binary-class` will contain a **`DEFCLASS`** and a `write-object` method just like the expansion of `define-binary-class`, but instead of a `read-object` method it'll contain a `read-value` method that looks like this:

```
(defmethod read-value ((type (eql 'id3-frame)) stream &key)
  (let ((id (read-value 'iso-8859-1-string stream :length 3))
        (size (read-value 'u3 stream)))
    (let ((object (make-instance (find-frame-class id) :id id :size size)))
      (read-object object stream)
      object)))
```

Since the expansions of `define-tagged-binary-class` and `define-binary-class` are going to be identical except for the read method, you can factor out the common bits into a helper macro, `define-generic-binary-class`, that accepts the read method as a parameter and interpolates it.

```
(defmacro define-generic-binary-class (name (&rest superclasses) slots read-method)
  (with-gensyms (objectvar streamvar)
    `(progn
      (eval-when (:compile-toplevel :load-toplevel :execute)
        (setf (get ',name 'slots) ',(mapcar #'first slots))
        (setf (get ',name 'superclasses) ',superclasses))

      (defclass ,name ,superclasses
        ,(mapcar #'slot->defclass-slot slots))

      ,read-method

      (defmethod write-object progn ((,objectvar ,name) ,streamvar)
        (declare (ignorable ,streamvar))
        (with-slots ,(new-class-all-slots slots superclasses) ,objectvar
          ,@(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```

Now you can define both `define-binary-class` and `define-tagged-binary-class` to expand into a call to `define-generic-binary-class`. Here's a new version of `define-binary-class` that generates the same code as the earlier version when it's fully expanded:

```
(defmacro define-binary-class (name (&rest superclasses) slots)
  (with-gensyms (objectvar streamvar)
    `(define-generic-binary-class ,name ,superclasses ,slots
```

```
      (defmethod read-object progn ((,objectvar ,name) ,streamvar)
        (declare (ignorable ,streamvar))
        (with-slots ,(new-class-all-slots slots superclasses) ,objectvar
          ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots))))))
```

And here's `define-tagged-binary-class` along with two new helper functions it uses:

```
(defmacro define-tagged-binary-class (name (&rest superclasses) slots &rest options)
  (with-gensyms (typevar objectvar streamvar)
    `(define-generic-binary-class ,name ,superclasses ,slots
      (defmethod read-value ((,typevar (eql ',name)) ,streamvar &key)
        (let* ,(mapcar #'(lambda (x) (slot->binding x streamvar)) slots)
          (let ((,objectvar
                 (make-instance
                  ,@(or (cdr (assoc :dispatch options))
                        (error "Must supply :dispatch form."))
                  ,@(mapcan #'slot->keyword-arg slots))))
            (read-object ,objectvar ,streamvar)
            ,objectvar))))))

(defun slot->binding (spec stream)
  (destructuring-bind (name (type &rest args)) (normalize-slot-spec spec)
    `(,name (read-value ',type ,stream ,@args))))

(defun slot->keyword-arg (spec)
  (let ((name (first spec)))
    `(,(as-keyword name) ,name)))
```

Primitive Binary Types

While `define-binary-class` and `define-tagged-binary-class` make it easy to define composite structures, you still have to write `read-value` and `write-value` methods for primitive data types by hand. You could decide to live with that, specifying that users of the library need to write appropriate methods on `read-value` and `write-value` to support the primitive types used by their binary classes.

However, rather than having to document how to write a suitable `read-value`/`write-value` pair, you can provide a macro to do it automatically. This also has the advantage of making the abstraction created by `define-binary-class` less leaky. Currently, `define-binary-class` depends on having methods on `read-value` and `write-value` defined in a particular way, but that's really just an implementation detail. By defining a macro that generates the `read-value` and `write-value` methods for primitive types, you hide those details behind an abstraction you control. If you decide later to change the implementation of `define-binary-class`, you can change your primitive-type-defining macro to meet the new requirements without requiring any changes to code that uses the binary data library.

So you should define one last macro, `define-binary-type`, that will generate `read-value` and `write-value` methods for reading values represented by instances of existing classes, rather than by classes defined with `define-binary-class`.

For a concrete example, consider a type used in the `id3-tag` class, a fixed-length string encoded in ISO-8859-1 characters. I'll assume, as I did earlier, that the native character encoding of your Lisp is ISO-8859-1 or a superset, so you can use **CODE-CHAR** and **CHAR-CODE** to translate bytes to characters and back.

As always, your goal is to write a macro that allows you to express only the essential information needed to generate the required code. In this case, there are four pieces of essential information: the name of the type, `iso-8859-1-string`; the **&key** parameters that should be accepted by the `read-value` and `write-value` methods, `length` in this case; the code for reading from a stream; and the code for writing to a stream. Here's an expression that contains those four pieces of information:

```
(define-binary-type iso-8859-1-string (length)
  (:reader (in)
    (let ((string (make-string length)))
      (dotimes (i length)
        (setf (char string i) (code-char (read-byte in))))
      string))
  (:writer (out string)
    (dotimes (i length)
      (write-byte (char-code (char string i)) out))))
```

Now you just need a macro that can take apart this form and put it back together in the form of two **DEFMETHOD**s wrapped in a **PROGN**. If you define the parameter list to `define-binary-type` like this:

```
(defmacro define-binary-type (name (&rest args) &body spec) ...
```

then within the macro the parameter `spec` will be a list containing the reader and writer definitions. You can then use **ASSOC** to extract the elements of `spec` using the tags `:reader` and `:writer` and then use **DESTRUCTURING-BIND** to take apart the **REST** of each element.[10]

From there it's just a matter of interpolating the extracted values into the backquoted templates of the `read-value` and `write-value` methods.

```
(defmacro define-binary-type (name (&rest args) &body spec)
  (with-gensyms (type)
    `(progn
      ,(destructuring-bind ((in) &body body) (rest (assoc :reader spec))
        `(defmethod read-value ((,type (eql ',name)) ,in &key ,@args)
          ,@body))
      ,(destructuring-bind ((out value) &body body) (rest (assoc :writer spec))
        `(defmethod write-value ((,type (eql ',name)) ,out ,value &key ,@args)
          ,@body)))))
```

Note how the backquoted templates are nested: the outermost template starts with the backquoted **PROGN** form. That template consists of the symbol **PROGN** and two comma-unquoted **DESTRUCTURING-BIND** expressions. Thus, the outer template is filled in by evaluating the **DESTRUCTURING-BIND** expressions and interpolating their values. Each **DESTRUCTURING-BIND** expression in turn contains another backquoted template, which is used to generate one of the method definitions to be interpolated in the outer template.

With this macro defined, the `define-binary-type` form given previously expands to this code:

```
(progn
  (defmethod read-value ((#:g1618 (eql 'iso-8859-1-string)) in &key length)
    (let ((string (make-string length)))
      (dotimes (i length)
        (setf (char string i) (code-char (read-byte in))))
      string))
  (defmethod write-value ((#:g1618 (eql 'iso-8859-1-string)) out string &key length)
    (dotimes (i length)
      (write-byte (char-code (char string i)) out))))
```

Of course, now that you've got this nice macro for defining binary types, it's tempting to make it do a bit more work. For now you should just make one small enhancement that will turn out to be pretty handy

when you start using this library to deal with actual formats such as ID3 tags.

ID3 tags, like many other binary formats, use lots of primitive types that are minor variations on a theme, such as unsigned integers in one-, two-, three-, and four-byte varieties. You could certainly define each of those types with `define-binary-type` as it stands. Or you could factor out the common algorithm for reading and writing *n*-byte unsigned integers into helper functions.

But suppose you had already defined a binary type, `unsigned-integer`, that accepts a `:bytes` parameter to specify how many bytes to read and write. Using that type, you could specify a slot representing a one-byte unsigned integer with a type specifier of `(unsigned-integer :bytes 1)`. But if a particular binary format specifies lots of slots of that type, it'd be nice to be able to easily define a new type--say, `u1`--that means the same thing. As it turns out, it's easy to change `define-binary-type` to support two forms, a long form consisting of a `:reader` and `:writer` pair and a short form that defines a new binary type in terms of an existing type. Using a short form `define-binary-type`, you can define `u1` like this:

```
(define-binary-type u1 () (unsigned-integer :bytes 1))
```

which will expand to this:

```
(progn
  (defmethod read-value ((#:g161887 (eql 'u1)) #:g161888 &key)
    (read-value 'unsigned-integer #:g161888 :bytes 1))
  (defmethod write-value ((#:g161887 (eql 'u1)) #:g161888 #:g161889 &key)
    (write-value 'unsigned-integer #:g161888 #:g161889 :bytes 1)))
```

To support both long- and short-form `define-binary-type` calls, you need to differentiate based on the value of the `spec` argument. If `spec`

is two items long, it represents a long-form call, and the two items should be the `:reader` and `:writer` specifications, which you extract as before. On the other hand, if it's only one item long, the one item should be a type specifier, which needs to be parsed differently. You can use **ECASE** to switch on the **LENGTH** of `spec` and then parse `spec` and generate an appropriate expansion for either the long form or the short form.

```
(defmacro define-binary-type (name (&rest args) &body spec)
  (ecase (length spec)
    (1
     (with-gensyms (type stream value)
       (destructuring-bind (derived-from &rest derived-args) (mklist (first spec))
         `(progn
            (defmethod read-value ((,type (eql ',name)) ,stream &key ,@args)
              (read-value ',derived-from ,stream ,@derived-args))
            (defmethod write-value ((,type (eql ',name)) ,stream ,value &key ,@args)
              (write-value ',derived-from ,stream ,value ,@derived-args))))))
    (2
     (with-gensyms (type)
       `(progn
          ,(destructuring-bind ((in) &body body) (rest (assoc :reader spec))
             `(defmethod read-value ((,type (eql ',name)) ,in &key ,@args)
                ,@body))
          ,(destructuring-bind ((out value) &body body) (rest (assoc :writer spec))
             `(defmethod write-value ((,type (eql ',name)) ,out ,value &key ,@args)
                ,@body)))))))
```

The Current Object Stack

One last bit of functionality you'll need in the next chapter is a way to get at the binary object being read or written while reading and writing. More generally, when reading or writing nested composite objects, it's useful to be able to get at any of the objects currently being read or written. Thanks to dynamic variables and `:around` methods, you can add this enhancement with about a dozen lines of code. To start, you should define a dynamic variable that will hold a stack of objects currently being read or written.

```
(defvar *in-progress-objects* nil)
```

Then you can define `:around` methods on `read-object` and `write-object` that push the object being read or written onto this variable before invoking **CALL-NEXT-METHOD**.

```
(defmethod read-object :around (object stream)
  (declare (ignore stream))
  (let ((*in-progress-objects* (cons object *in-progress-objects*)))
    (call-next-method)))

(defmethod write-object :around (object stream)
  (declare (ignore stream))
  (let ((*in-progress-objects* (cons object *in-progress-objects*)))
    (call-next-method)))
```

Note how you rebind `*in-progress-objects*` to a list with a new item on the front rather than assigning it a new value. This way, at the end of the **LET**, after **CALL-NEXT-METHOD** returns, the old value of `*in-progress-objects*` will be restored, effectively popping the object of the stack.

With those two methods defined, you can provide two convenience functions for getting at specific objects in the in-progress stack. The

function `current-binary-object` will return the head of the stack, the object whose `read-object` or `write-object` method was invoked most recently. The other, `parent-of-type`, takes an argument that should be the name of a binary object class and returns the most recently pushed object of that type, using the **`TYPEP`** function that tests whether a given object is an instance of a particular type.

```
(defun current-binary-object () (first *in-progress-objects*))

(defun parent-of-type (type)
  (find-if #'(lambda (x) (typep x type)) *in-progress-objects*))
```

These two functions can be used in any code that will be called within the dynamic extent of a `read-object` or `write-object` call. You'll see one example of how `current-binary-object` can be used in the next chapter.[11]

Now you have all the tools you need to tackle an ID3 parsing library, so you're ready to move onto the next chapter where you'll do just that.

---

[1]In ASCII, the first 32 characters are nonprinting *control characters* originally used to control the behavior of a Teletype machine, causing it to do such things as sound the bell, back up one character, move to a new line, and move the carriage to the beginning of the line. Of these 32 control characters, only three, the newline, carriage return, and horizontal tab, are typically found in text files.

[2]Some binary file formats *are* in-memory data structures--on many operating systems it's possible to map a file into memory, and low-level languages such as C can then treat the region of memory containing the contents of the file just like any other memory; data written to that area of memory is saved to the underlying file when it's unmapped. However, these formats are platform-dependent since the in-memory representation of even such simple data types as integers depends on the hardware on which the program is running. Thus, any file format that's intended to be portable must define a canonical representation for all

the data types it uses that can be mapped to the actual in-memory data representation on a particular kind of machine or in a particular language.

[3]The term *big-endian* and its opposite, *little-endian*, borrowed from Jonathan Swift's *Gulliver's Travels*, refer to the way a multibyte number is represented in an ordered sequence of bytes such as in memory or in a file. For instance, the number 43981, or `abcd` in hex, represented as a 16-bit quantity, consists of two bytes, `ab` and `cd`. It doesn't matter to a computer in what order these two bytes are stored as long as everybody agrees. Of course, whenever there's an arbitrary choice to be made between two equally good options, the one thing you can be sure of is that everybody is not going to agree. For more than you ever wanted to know about it, and to see where the terms *big-endian* and *little-endian* were first applied in this fashion, read "On Holy Wars and a Plea for Peace" by Danny Cohen, available at `http://khavrinen.lcs.mit.edu/wollman/ien-137.txt`.

[4]**LDB** and **DPB**, a related function, were named after the DEC PDP-10 assembly functions that did essentially the same thing. Both functions operate on integers as if they were represented using twos-complement format, regardless of the internal representation used by a particular Common Lisp implementation.

[5]Common Lisp also provides functions for shifting and masking the bits of integers in a way that may be more familiar to C and Java programmers. For instance, you could write `read-u2` yet a third way, using those functions, like this:

```
(defun read-u2 (in)
  (logior (ash (read-byte in) 8) (read-byte in)))
```

which would be roughly equivalent to this Java method:

```
public int readU2 (InputStream in) throws IOException {
  return (in.read() << 8) | (in.read());
}
```

The names **LOGIOR** and **ASH** are short for *LOGical Inclusive OR* and *Arithmetic SHift*. **ASH** shifts an integer a given number of bits to the left when its second argument is positive or to the right if the second argument is negative. **LOGIOR** combines integers by logically *or*ing each bit. Another function, **LOGAND**, performs a bitwise *and*, which can be used to mask off certain bits. However, for the kinds of bit twiddling you'll need to do in this chapter and the next, **LDB** and **BYTE** will be both more convenient and more idiomatic Common Lisp style.

[6]Originally, UTF-8 was designed to represent a 31-bit character code and used up to six bytes per code point. However, the maximum Unicode code point is `#x10ffff`, so a UTF-8 encoding of Unicode requires at most four bytes per code point.

[7]If you need to parse a file format that uses other character codes, or if you need to parse files containing arbitrary Unicode strings using a non-Unicode-Common-Lisp implementation, you can always represent such strings in memory as vectors of integer code points. They won't be Lisp strings, so you won't be able to manipulate or compare them with the string functions, but you'll still be able to do anything with them that you can with arbitrary vectors.

[8]Unfortunately, the language itself doesn't always provide a good model in this respect: the macro **DEFSTRUCT**, which I don't discuss since it has largely been superseded by **DEFCLASS**, generates functions with names that it generates based on the name of the structure it's given. **DEFSTRUCT**'s bad example leads many new macro writers astray.

[9]Technically there's no possibility of `type` or `object` conflicting with slot names--at worst they'd be shadowed within the **WITH-SLOTS** form. But it doesn't hurt anything to simply **GENSYM** all local variable names used within a macro template.

[10]Using **ASSOC** to extract the `:reader` and `:writer` elements of `spec` allows users of `define-binary-type` to include the elements in either order; if you required the `:reader` element to be always be first, you could then have used `(rest (first spec))` to extract the reader and `(rest (second spec))` to extract the writer. However, as long as you require the `:reader` and `:writer` keywords to improve the readability of `define-binary-type` forms, you might as well use them to extract the correct data.

[11]The ID3 format doesn't require the `parent-of-type` function since it's a relatively flat structure. This function comes into its own when you need to parse a format made up of many deeply nested structures whose parsing depends on information stored in higher-level structures. For example, in the Java class file format, the top-level class file structure contains a *constant pool* that maps numeric values used in other substructures within the class file to constant values that are needed while parsing those substructures. If you were writing a class file parser, you could use `parent-of-type` in the code that reads and writes those substructures to get at the top-level class file object and from there to the constant pool.

Copyright © 2003-2005, Peter Seibel

25. Practical: An ID3 Parser

With a library for parsing binary data, you're ready to write some code for reading and writing an actual binary format, that of ID3 tags. ID3 tags are used to embed metadata in MP3 audio files. Dealing with ID3 tags will be a good test of the binary data library because the ID3 format is a true real-world format--a mix of engineering trade-offs and idiosyncratic design choices that does, whatever else might be said about it, get the job done. In case you missed the file-sharing revolution, here's a quick overview of what ID3 tags are and how they relate to MP3 files.

MP3, also known as MPEG Audio Layer 3, is a format for storing compressed audio data, designed by researchers at Fraunhofer IIS and standardized by the Moving Picture Experts Group, a joint committee of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). However, the MP3 format, by itself, defines only how to store audio data. That's fine as long as all your MP3 files are managed by a single application that can store metadata externally and keep track of which metadata goes with which files. However, when people started passing around individual MP3 files on the Internet, via file-sharing systems such as Napster, they soon discovered they needed a way to embed metadata in the MP3 files themselves.

Because the MP3 standard was already codified and a fair bit of software and hardware had already been written that knew how to decode the existing MP3 format, any scheme for embedding

information in an MP3 file would have to be invisible to MP3 decoders. Enter ID3.

The original ID3 format, invented by programmer Eric Kemp, consisted of 128 bytes stuck on the end of an MP3 file where it'd be ignored by most MP3 software. It consisted of four 30-character fields, one each for the song title, the album title, the artist name, and a comment; a four-byte year field; and a one-byte genre code. Kemp provided standard meanings for the first 80 genre codes. Nullsoft, the makers of Winamp, a popular MP3 player, later supplemented this list with another 60 or so genres.

This format was easy to parse but obviously quite limited. It had no way to encode names longer than 30 characters; it was limited to 256 genres, and the meaning of the genre codes had to be agreed upon by all users of ID3-aware software. There wasn't even a way to encode the CD track number of a particular MP3 file until another programmer, Michael Mutschler, proposed embedding the track number in the comment field, separated from the rest of the comment by a null byte, so existing ID3 software, which tended to read up to the first null in each of the text fields, would ignore it. Kemp's version is now called ID3v1, and Mutschler's is ID3v1.1.

Limited as they were, the version 1 proposals were at least a partial solution to the metadata problem, so they were adopted by many MP3 ripping programs (which had to put the ID3 tag into the MP3 files) and MP3 players (which would extract the information in the ID3 tag to display to the user).[1]

By 1998, however, the limitations were really becoming annoying, and a new group, led by Martin Nilsson, started work on a completely new tagging scheme, which came to be called ID3v2. The ID3v2 format is extremely flexible, allowing for many kinds of information to be included, with almost no length limitations. It also takes advantage of certain details of the MP3 format to allow ID3v2 tags to be placed at the beginning of an MP3 file.

ID3v2 tags are, however, more of a challenge to parse than version 1 tags. In this chapter, you'll use the binary data parsing library from the previous chapter to develop code that can read and write ID3v2 tags. Or at least you'll make a reasonable start--where ID3v1 was too simple, ID3v2 is baroque to the point of being completely overengineered. Implementing every nook and cranny of the specification, especially if you want to support all three versions that have been specified, would be a fair bit of work. However, you can ignore many of the features in those specifications since they're rarely used "in the wild." For starters, you can ignore, for now, a whole version, 2.4, since it has not been widely adopted and mostly just adds more needless flexibility compared to version 2.3. I'll focus on versions 2.2 and 2.3 because they're both widely used and are different enough from each other to keep things interesting.

Structure of an ID3v2 Tag

Before you can start cutting code, you'll need to be familiar with the overall structure of an ID3v2 tag. A tag starts with a header containing information about the tag as a whole. The first three bytes of the header encode the string "ID3" in ISO-8859-1 characters. In other words, they're the bytes 73, 68, and 51. Then comes two bytes that encode the major version and revision of the ID3 specification to which the tag purports to conform. They're followed by a single byte whose individual bits are treated as flags. The meanings of the individual flags depend on the version of the spec. Some of the flags can affect the way the rest of the tag is parsed. The "major version" is actually used to record the minor version of the spec, while the "revision" is the subminor version of the spec. Thus, the "major version" field for a tag conforming to the 2.3.0 spec is 3. The revision field is always zero since each new ID3v2 spec has bumped the minor version, leaving the subminor version at zero. The value stored in the major version field of the tag has, as you'll see, a dramatic effect on how you'll parse the rest of the tag.

The last field in the tag header is an integer, encoded in four bytes but using only seven bits from each byte, that gives the total size of the tag, not counting the header. In version 2.3 tags, the header may be followed by several *extended header* fields; otherwise, the remainder of the tag data is divided into *frames*. Different types of frames store different kinds of information, from simple textual information, such as the song name, to embedded images. Each frame starts with a header containing a string identifier and a size. In

version 2.3, the frame header also contains two bytes worth of flags and, depending on the value of one the flags, an optional one-byte code indicating how the rest of the frame is encrypted.

Frames are a perfect example of a tagged data structure--to know how to parse the body of a frame, you need to read the header and use the identifier to determine what kind of frame you're reading.

The ID3 tag header contains no direct indication of how many frames are in a tag--the tag header tells you how big the tag is, but since many frames are variable length, the only way to find out how many frames the tag contains is to read the frame data. Also, the size given in the tag header may be larger than the actual number of bytes of frame data; the frames may be followed with enough null bytes to pad the tag out to the specified size. This makes it possible for tag editors to modify a tag without having to rewrite the whole MP3 file.[2]

So, the main issues you have to deal with are reading the ID3 header; determining whether you're reading a version 2.2 or 2.3 tag; and reading the frame data, stopping either when you've read the complete tag or when you've hit the padding bytes.

Defining a Package

Like the other libraries you've developed so far, the code you'll write in this chapter is worth putting in its own package. You'll need to refer to functions from both the binary data and pathname libraries developed in Chapters 24 and 15 and will also want to export the names of the functions that make up the public API to this package. The following package definition does all that:

```
(defpackage :com.gigamonkeys.id3v2
  (:use :common-lisp
        :com.gigamonkeys.binary-data
        :com.gigamonkeys.pathnames)
  (:export
   :read-id3
   :mp3-p
   :id3-p
   :album
   :composer
   :genre
   :encoding-program
   :artist
   :part-of-set
   :track
   :song
   :year
   :size
   :translated-genre))
```

As usual, you can, and probably should, change the `com.gigamonkeys` part of the package name to your own domain.

Integer Types

You can start by defining binary types for reading and writing several of the primitive types used by the ID3 format, various sizes of unsigned integers, and four kinds of strings.

ID3 uses unsigned integers encoded in one, two, three, and four bytes. If you first write a general `unsigned-integer` binary type that takes the number of bytes to read as an argument, you can then use the short form of `define-binary-type` to define the specific types. The general `unsigned-integer` type looks like this:

```
(define-binary-type unsigned-integer (bytes)
  (:reader (in)
    (loop with value = 0
       for low-bit downfrom (* 8 (1- bytes)) to 0 by 8 do
         (setf (ldb (byte 8 low-bit) value) (read-byte in))
       finally (return value)))
  (:writer (out value)
    (loop for low-bit downfrom (* 8 (1- bytes)) to 0 by 8
       do (write-byte (ldb (byte 8 low-bit) value) out))))
```

Now you can use the short form of `define-binary-type` to define one type for each size of integer used in the ID3 format like this:

```
(define-binary-type u1 () (unsigned-integer :bytes 1))
(define-binary-type u2 () (unsigned-integer :bytes 2))
(define-binary-type u3 () (unsigned-integer :bytes 3))
(define-binary-type u4 () (unsigned-integer :bytes 4))
```

Another type you'll need to be able to read and write is the 28-bit value used in the header. This size is encoded using 28 bits rather than a multiple of 8, such as 32 bits, because an ID3 tag can't contain the byte `#xff` followed by a byte with the top 3 bits on because that pattern has a special meaning to MP3 decoders. None of the other fields in the ID3 header could possibly contain such a

byte sequence, but if you encoded the tag size as a regular `unsigned-integer`, it might. To avoid that possibility, the size is encoded using only the bottom seven bits of each byte, with the top bit always zero.[3]

Thus, it can be read and written a lot like an `unsigned-integer` except the size of the byte specifier you pass to **LDB** should be seven rather than eight. This similarity suggests that if you add a parameter, `bits-per-byte`, to the existing `unsigned-integer` binary type, you could then define a new type, `id3-tag-size`, using a short-form `define-binary-type`. The new version of `unsigned-integer` is just like the old version except with `bits-per-byte` used everywhere the old version hardwired the number eight. It looks like this:

```
(define-binary-type unsigned-integer (bytes bits-per-byte)
  (:reader (in)
    (loop with value = 0
       for low-bit downfrom (* bits-per-byte (1- bytes)) to 0 by bits-per-byte do
         (setf (ldb (byte bits-per-byte low-bit) value) (read-byte in))
       finally (return value)))
  (:writer (out value)
    (loop for low-bit downfrom (* bits-per-byte (1- bytes)) to 0 by bits-per-byte
       do (write-byte (ldb (byte bits-per-byte low-bit) value) out))))
```

The definition of `id3-tag-size` is then trivial.

```
(define-binary-type id3-tag-size () (unsigned-integer :bytes 4 :bits-per-byte 7))
```

You'll also have to change the definitions of `u1` through `u4` to specify eight bits per byte like this:

```
(define-binary-type u1 () (unsigned-integer :bytes 1 :bits-per-byte 8))
(define-binary-type u2 () (unsigned-integer :bytes 2 :bits-per-byte 8))
(define-binary-type u3 () (unsigned-integer :bytes 3 :bits-per-byte 8))
(define-binary-type u4 () (unsigned-integer :bytes 4 :bits-per-byte 8))
```

String Types

The other kinds of primitive types that are ubiquitous in the ID3 format are strings. In the previous chapter I discussed some of the issues you have to consider when dealing with strings in binary files, such as the difference between character codes and character encodings.

ID3 uses two different character codes, ISO 8859-1 and Unicode. ISO 8859-1, also known as Latin-1, is an eight-bit character code that extends ASCII with characters used by the languages of Western Europe. In other words, the code points from 0-127 map to the same characters in ASCII and ISO 8859-1, but ISO 8859-1 also provides mappings for code points up to 255. Unicode is a character code designed to provide a code point for virtually every character of all the world's languages. Unicode is a superset of ISO 8859-1 in the same way that ISO 8859-1 is a superset of ASCII--the code points from 0-255 map to the same characters in both ISO 8859-1 and Unicode. (Thus, Unicode is also a superset of ASCII.)

Since ISO 8859-1 is an eight-bit character code, it's encoded using one byte per character. For Unicode strings, ID3 uses the UCS-2 encoding with a leading *byte order mark*.[4] I'll discuss what a byte order mark is in a moment.

Reading and writing these two encodings isn't a problem--it's just a question of reading and writing unsigned integers in various formats, and you just finished writing the code to do that. The trick is how you translate those numeric values to Lisp character objects.

The Lisp implementation you're using probably uses either Unicode or ISO 8859-1 as its internal character code. And since all the values from 0-255 map to the same characters in both ISO 8859-1 and Unicode, you can use Lisp's `CODE-CHAR` and `CHAR-CODE` functions to translate those values in both character codes. However, if your Lisp supports only ISO 8859-1, then you'll be able to represent only the first 255 Unicode characters as Lisp characters. In other words, in such a Lisp implementation, if you try to process an ID3 tag that uses Unicode strings and if any of those strings contain characters with code points higher than 255, you'll get an error when you try to translate the code point to a Lisp character. For now I'll assume either you're using a

Unicode-based Lisp or you won't process any files containing characters outside the ISO 8859-1 range.

The other issue with encoding strings is how to know how many bytes to interpret as character data. ID3 uses two strategies I mentioned in the previous chapter--some strings are terminated with a null character, while other strings occur in positions where you can determine the number of bytes to read, either because the string at that position is always the same length or because the string is at the end of a composite structure whose overall size you know. Note, however, that the number of bytes isn't necessarily the same as the number of characters in the string.

Putting all these variations together, the ID3 format uses four ways to read and write strings--two characters crossed with two ways of delimiting the string data.

Obviously, much of the logic of reading and writing strings will be quite similar. So, you can start by defining two binary types, one for reading strings of a specific length (in characters) and another for reading terminated strings. Both types take advantage of that the type argument to `read-value` and `write-value` is just another piece of data; you can make the type of character to read a parameter of these types. This is a technique you'll use quite a few times in this chapter.

```
(define-binary-type generic-string (length character-type)
  (:reader (in)
    (let ((string (make-string length)))
      (dotimes (i length)
        (setf (char string i) (read-value character-type in)))
      string))
  (:writer (out string)
    (dotimes (i length)
      (write-value character-type out (char string i)))))

(define-binary-type generic-terminated-string (terminator character-type)
  (:reader (in)
    (with-output-to-string (s)
      (loop for char = (read-value character-type in)
            until (char= char terminator) do (write-char char s))))
  (:writer (out string)
    (loop for char across string
          do (write-value character-type out char)
          finally (write-value character-type out terminator))))
```

With these types available, there's not much to reading ISO 8859-1 strings. Because the `character-type` argument you pass to `read-value` and `write-value` of a `generic-string` must be the name of a binary type, you need to define an `iso-8859-1-char` binary type. This also gives you a good place to put a bit of sanity checking on the code points of characters you read and write.

```
(define-binary-type iso-8859-1-char ()
  (:reader (in)
```

```
    (let ((code (read-byte in)))
      (or (code-char code)
          (error "Character code ~d not supported" code))))
  (:writer (out char)
    (let ((code (char-code char)))
      (if (<= 0 code #xff)
          (write-byte code out)
          (error "Illegal character for iso-8859-1 encoding: character: ~c with code: ~d" char code)))))
```

Now defining the ISO 8859-1 string types is trivial using the short form of `define-binary-type` as follows:

```
(define-binary-type iso-8859-1-string (length)
  (generic-string :length length :character-type 'iso-8859-1-char))

(define-binary-type iso-8859-1-terminated-string (terminator)
  (generic-terminated-string :terminator terminator :character-type 'iso-8859-1-char))
```

Reading UCS-2 strings is only slightly more complex. The complexity arises because you can encode a UCS-2 code point in two ways: most significant byte first (big-endian) or least significant byte first (little-endian). UCS-2 strings therefore start with two extra bytes, called the *byte order mark*, made up of the numeric value `#xfeff` encoded in either big-endian form or little-endian form. When reading a UCS-2 string, you read the byte order mark and then, depending on its value, read either big-endian or little-endian characters. Thus, you'll need two different UCS-2 character types. But you need only one version of the sanity-checking code, so you can define a parameterized binary type like this:

```
(define-binary-type ucs-2-char (swap)
  (:reader (in)
    (let ((code (read-value 'u2 in)))
      (when swap (setf code (swap-bytes code)))
      (or (code-char code) (error "Character code ~d not supported" code))))
  (:writer (out char)
    (let ((code (char-code char)))
      (unless (<= 0 code #xffff)
        (error "Illegal character for ucs-2 encoding: ~c with char-code: ~d" char code))
      (when swap (setf code (swap-bytes code)))
      (write-value 'u2 out code))))
```

where the `swap-bytes` function can be defined as follows, taking advantage of **LDB** being **SETF**able and thus **ROTATEF**able:

```
(defun swap-bytes (code)
  (assert (<= code #xffff))
  (rotatef (ldb (byte 8 0) code) (ldb (byte 8 8) code))
  code)
```

Using `ucs-2-char`, you can define two character types that will be used as the `character-type` arguments to the generic string functions.

```
(define-binary-type ucs-2-char-big-endian () (ucs-2-char :swap nil))

(define-binary-type ucs-2-char-little-endian () (ucs-2-char :swap t))
```

Then you need a function that returns the name of the character type to use based on the value of the byte order mark.

```
(defun ucs-2-char-type (byte-order-mark)
  (ecase byte-order-mark
    (#xfeff 'ucs-2-char-big-endian)
    (#xfffe 'ucs-2-char-little-endian)))
```

Now you can define length- and terminator-delimited string types for UCS-2-encoded strings that read the byte order mark and use it to determine which variant of UCS-2 character to pass as the `character-type` argument to `read-value` and `write-value`. The only other wrinkle is that you need to translate the `length` argument, which is a number of bytes, to the number of characters to read, accounting for the byte order mark.

```
(define-binary-type ucs-2-string (length)
  (:reader (in)
    (let ((byte-order-mark (read-value 'u2 in))
          (characters (1- (/ length 2))))
      (read-value
       'generic-string in
       :length characters
       :character-type (ucs-2-char-type byte-order-mark))))
  (:writer (out string)
    (write-value 'u2 out #xfeff)
    (write-value
     'generic-string out string
     :length (length string)
     :character-type (ucs-2-char-type #xfeff))))

(define-binary-type ucs-2-terminated-string (terminator)
  (:reader (in)
    (let ((byte-order-mark (read-value 'u2 in)))
      (read-value
       'generic-terminated-string in
       :terminator terminator
       :character-type (ucs-2-char-type byte-order-mark))))
  (:writer (out string)
    (write-value 'u2 out #xfeff)
    (write-value
     'generic-terminated-string out string
     :terminator terminator
     :character-type (ucs-2-char-type #xfeff))))
```

ID3 Tag Header

With the basic primitive types done, you're ready to switch to a high-level view and start defining binary classes to represent first the ID3 tag as a whole and then the individual frames.

If you turn first to the ID3v2.2 specification, you'll see that the basic structure of the tag is this header:

```
ID3/file identifier      "ID3"
ID3 version              $02 00
ID3 flags                %xx000000
ID3 size             4 * %0xxxxxxx
```

followed by frame data and padding. Since you've already defined binary types to read and write all the fields in the header, defining a class that can read the header of an ID3 tag is just a matter of putting them together.

```
(define-binary-class id3-tag ()
  ((identifier     (iso-8859-1-string :length 3))
   (major-version  u1)
   (revision       u1)
   (flags          u1)
   (size           id3-tag-size)))
```

If you have some MP3 files lying around, you can test this much of the code and also see what version of ID3 tags your MP3s contain. First you can write a function that reads an `id3-tag`, as just defined, from the beginning of a file. Be aware, however, that ID3 tags aren't required to appear at the beginning of a file, though these days they almost always do. To find an ID3 tag elsewhere in a file, you can scan the file looking for the sequence of bytes 73, 68, 51 (in other words, the string "ID3").[5] For now you can probably get away with assuming the tags are the first thing in the file.

```
(defun read-id3 (file)
  (with-open-file (in file :element-type '(unsigned-byte 8))
    (read-value 'id3-tag in)))
```

On top of this function you can build a function that takes a filename and prints the information in the tag header along with the name of the file.

```
(defun show-tag-header (file)
  (with-slots (identifier major-version revision flags size) (read-id3 file)
    (format t "~a ~d.~d ~8,'0b ~d bytes -- ~a~%"
            identifier major-version revision flags size (enough-namestring file))))
```

It prints output that looks like this:

```
ID3V2> (show-tag-header "/usr2/mp3/Kitka/Wintersongs/02 Byla Cesta.mp3")
ID3 2.0 00000000 2165 bytes -- Kitka/Wintersongs/02 Byla Cesta.mp3
NIL
```

Of course, to determine what versions of ID3 are most common in your MP3 library, it'd be handier to have a function that returns a summary of all the MP3 files under a given directory. You can write one easily enough using the `walk-directory` function defined in Chapter 15. First define a helper function that tests whether a given filename has an `mp3` extension.

```
(defun mp3-p (file)
  (and
   (not (directory-pathname-p file))
   (string-equal "mp3" (pathname-type file))))
```

Then you can combine `show-tag-header` and `mp3-p` with `walk-directory` to print a summary of the ID3 header in each file under a given directory.

```
(defun show-tag-headers (dir)
  (walk-directory dir #'show-tag-header :test #'mp3-p))
```

However, if you have a lot of MP3s, you may just want a count of how many ID3 tags of each version you have in your MP3 collection. To

get that information, you might write a function like this:

```
(defun count-versions (dir)
  (let ((versions (mapcar #'(lambda (x) (cons x 0)) '(2 3 4))))
    (flet ((count-version (file)
             (incf (cdr (assoc (major-version (read-id3 file)) versions)))))
      (walk-directory dir #'count-version :test #'mp3-p))
    versions))
```

Another function you'll need in Chapter 29 is one that tests whether a file actually starts with an ID3 tag, which you can define like this:

```
(defun id3-p (file)
  (with-open-file (in file :element-type '(unsigned-byte 8))
    (string= "ID3" (read-value 'iso-8859-1-string in :length 3))))
```

ID3 Frames

As I discussed earlier, the bulk of an ID3 tag is divided into frames. Each frame has a structure similar to that of the tag as a whole. Each frame starts with a header indicating what kind of frame it is and the size of the frame in bytes. The structure of the frame header changed slightly between version 2.2 and version 2.3 of the ID3 format, and eventually you'll have to deal with both forms. To start, you can focus on parsing version 2.2 frames.

The header of a 2.2 frame consists of three bytes that encode a three-character ISO 8859-1 string followed by a three-byte unsigned integer, which specifies the size of the frame in bytes, excluding the six-byte header. The string identifies what type of frame it is, which determines how you parse the data following the size. This is exactly the kind of situation for which you defined the `define-tagged-binary-class` macro. You can define a tagged class that reads the frame header and then dispatches to the appropriate concrete class using a function that maps IDs to a class names.

```
(define-tagged-binary-class id3-frame ()
  ((id (iso-8859-1-string :length 3))
   (size u3))
  (:dispatch (find-frame-class id)))
```

Now you're ready to start implementing concrete frame classes. However, the specification defines quite a few--63 in version 2.2 and even more in later specs. Even considering frame types that share a common structure to be equivalent, you'll still find 24 unique frame types in version 2.2. But only a few of these are used "in the wild." So rather than immediately setting to work defining classes for each

of the frame types, you can start by writing a generic frame class that lets you read the frames in a tag without parsing the data within the frames themselves. This will give you a way to find out what frames are actually present in the MP3s you want to process. You'll need this class eventually anyway because the specification allows for experimental frames that you'll need to be able to read without parsing.

Since the size field of the frame header tells you exactly how many bytes long the frame is, you can define a `generic-frame` class that extends `id3-frame` and adds a single field, `data`, that will hold an array of bytes.

```
(define-binary-class generic-frame (id3-frame)
  ((data (raw-bytes :size size))))
```

The type of the data field, `raw-bytes`, just needs to hold an array of bytes. You can define it like this:

```
(define-binary-type raw-bytes (size)
  (:reader (in)
    (let ((buf (make-array size :element-type '(unsigned-byte 8))))
      (read-sequence buf in)
      buf))
  (:writer (out buf)
    (write-sequence buf out)))
```

For the time being, you'll want all frames to be read as `generic-frame`s, so you can define the `find-frame-class` function used in `id3-frame`'s `:dispatch` expression to always return `generic-frame`, regardless of the frame's `id`.

```
(defun find-frame-class (id)
  (declare (ignore id))
  'generic-frame)
```

Now you need to modify `id3-tag` so it'll read frames after the header fields. There's only one tricky bit to reading the frame data: although the tag header tells you how many bytes long the tag is, that number includes the padding that can follow the frame data. Since the tag header doesn't tell you how many frames the tag contains, the only way to tell when you've hit the padding is to look for a null byte where you'd expect a frame identifier.

To handle this, you can define a binary type, `id3-frames`, that will be responsible for reading the remainder of a tag, creating frame objects to represent all the frames it finds, and then skipping over any padding. This type will take as a parameter the tag size, which it can use to avoid reading past the end of the tag. But the reading code will also need to detect the beginning of the padding that can follow the tag's frame data. Rather than calling `read-value` directly in `id3-frames :reader`, you should use a function `read-frame`, which you'll define to return `NIL` when it detects padding, otherwise returning an `id3-frame` object read using `read-value`. Assuming you define `read-frame` so it reads only one byte past the end of the last frame in order to detect the start of the padding, you can define the `id3-frames` binary type like this:

```
(define-binary-type id3-frames (tag-size)
  (:reader (in)
    (loop with to-read = tag-size
          while (plusp to-read)
          for frame = (read-frame in)
          while frame
          do (decf to-read (+ 6 (size frame)))
          collect frame
          finally (loop repeat (1- to-read) do (read-byte in))))
  (:writer (out frames)
    (loop with to-write = tag-size
          for frame in frames
          do (write-value 'id3-frame out frame)
          (decf to-write (+ 6 (size frame)))
          finally (loop repeat to-write do (write-byte 0 out)))))
```

You can use this type to add a `frames` slot to `id3-tag`.

```
(define-binary-class id3-tag ()
  ((identifier     (iso-8859-1-string :length 3))
   (major-version  u1)
   (revision       u1)
   (flags          u1)
   (size           id3-tag-size)
   (frames         (id3-frames :tag-size size))))
```

Detecting Tag Padding

Now all that remains is to implement `read-frame`. This is a bit tricky since the code that actually reads bytes from the stream is several layers down from `read-frame`.

What you'd really like to do in `read-frame` is read one byte and return **`NIL`** if it's a null and otherwise read a frame with `read-value`. Unfortunately, if you read the byte in `read-frame`, then it won't be available to be read by `read-value`.[6]

It turns out this is a perfect opportunity to use the condition system--you can check for null bytes in the low-level code that reads from the stream and signal a condition when you read a null; `read-frame` can then handle the condition by unwinding the stack before more bytes are read. In addition to turning out to be a tidy solution to the problem of detecting the start of the tag's padding, this is also an example of how you can use conditions for purposes other than handling errors.

You can start by defining a condition type to be signaled by the low-level code and handled by the high-level code. This condition doesn't need any slots--you just need a distinct class of condition so you know no other code will be signaling or handling it.

```
(define-condition in-padding () ())
```

Next you need to define a binary type whose `:reader` reads a given number of bytes, first reading a single byte and signaling an `in-padding` condition if the byte is null and otherwise reading the

remaining bytes as an `iso-8859-1-string` and combining it with the first byte read.

```
(define-binary-type frame-id (length)
  (:reader (in)
    (let ((first-byte (read-byte in)))
      (when (= first-byte 0) (signal 'in-padding))
      (let ((rest (read-value 'iso-8859-1-string in :length (1- length))))
        (concatenate
         'string (string (code-char first-byte)) rest))))
  (:writer (out id)
    (write-value 'iso-8859-1-string out id :length length)))
```

If you redefine `id3-frame` to make the type of its `id` slot `frame-id` instead of `iso-8859-1-string`, the condition will be signaled whenever `id3-frame`'s `read-value` method reads a null byte instead of the beginning of a frame.

```
(define-tagged-binary-class id3-frame ()
  ((id (frame-id :length 3))
   (size u3))
  (:dispatch (find-frame-class id)))
```

Now all `read-frame` has to do is wrap a call to `read-value` in a **`HANDLER-CASE`** that handles the `in-padding` condition by returning **`NIL`**.

```
(defun read-frame (in)
  (handler-case (read-value 'id3-frame in)
    (in-padding () nil)))
```

With `read-frame` defined, you can now read a complete version 2.2 ID3 tag, representing frames with instances of `generic-frame`. In the "What Frames Do You Actually Need?" section, you'll do some experiments at the REPL to determine what frame classes you need to implement. But first let's add support for version 2.3 ID3 tags.

Supporting Multiple Versions of ID3

Currently, `id3-tag` is defined using `define-binary-class`, but if you want to support multiple versions of ID3, it makes more sense to use a `define-tagged-binary-class` that dispatches on the `major-version` value. As it turns out, all versions of ID3v2 have the same structure up to the size field. So, you can define a tagged binary class like the following that defines this basic structure and then dispatches to the appropriate version-specific subclass:

```
(define-tagged-binary-class id3-tag ()
  ((identifier     (iso-8859-1-string :length 3))
   (major-version  u1)
   (revision       u1)
   (flags          u1)
   (size           id3-tag-size))
  (:dispatch
   (ecase major-version
     (2 'id3v2.2-tag)
     (3 'id3v2.3-tag))))
```

Version 2.2 and version 2.3 tags differ in two ways. First, the header of a version 2.3 tag may be extended with up to four optional extended header fields, as determined by values in the flags field. Second, the frame format changed between version 2.2 and version 2.3, which means you'll have to use different classes to represent version 2.2 frames and the corresponding version 2.3 frames.

Since the new `id3-tag` class is based on the one you originally wrote to represent version 2.2 tags, it's not surprising that the new `id3v2.2-tag` class is trivial, inheriting most of its slots from the new `id3-tag` class and adding the one missing slot, `frames`. Because version 2.2 and version 2.3 tags use different frame formats, you'll have to change the `id3-frames` type to be parameterized with the

type of frame to read. For now, assume you'll do that and add a `:frame-type` argument to the `id3-frames` type descriptor like this:

```
(define-binary-class id3v2.2-tag (id3-tag)
  ((frames (id3-frames :tag-size size :frame-type 'id3v2.2-frame))))
```

The `id3v2.3-tag` class is slightly more complex because of the optional fields. The first three of the four optional fields are included when the sixth bit in `flags` is set. They're a four- byte integer specifying the size of the extended header, two bytes worth of flags, and another four-byte integer specifying how many bytes of padding are included in the tag.[7] The fourth optional field, included when the fifteenth bit of the extended header flags is set, is a four-byte cyclic redundancy check (CRC) of the rest of the tag.

The binary data library doesn't provide any special support for optional fields in a binary class, but it turns out that regular parameterized binary types are sufficient. You can define a type parameterized with the name of a type and a value that indicates whether a value of that type should actually be read or written.

```
(define-binary-type optional (type if)
  (:reader (in)
    (when if (read-value type in)))
  (:writer (out value)
    (when if (write-value type out value))))
```

Using `if` as the parameter name looks a bit strange in that code, but it makes the `optional` type descriptors quite readable. For instance, here's the definition of `id3v2.3-tag` using `optional` slots:

```
(define-binary-class id3v2.3-tag (id3-tag)
  ((extended-header-size (optional :type 'u4 :if (extended-p flags)))
   (extra-flags          (optional :type 'u2 :if (extended-p flags)))
   (padding-size         (optional :type 'u4 :if (extended-p flags)))
   (crc                  (optional :type 'u4 :if (crc-p flags extra-flags)))
   (frames               (id3-frames :tag-size size :frame-type 'id3v2.3-frame))))
```

where `extended-p` and `crc-p` are helper functions that test the appropriate bit of the flags value they're passed. To test whether an individual bit of an integer is set, you can use **`LOGBITP`**, another bit-twiddling function. It takes an index and an integer and returns true if the specified bit is set in the integer.

```
(defun extended-p (flags) (logbitp 6 flags))

(defun crc-p (flags extra-flags)
  (and (extended-p flags) (logbitp 15 extra-flags)))
```

As in the version 2.2 tag class, the frames slot is defined to be of type `id3-frames`, passing the name of the frame type as a parameter. You do, however, need to make a few small changes to `id3-frames` and `read-frame` to support the extra `frame-type` parameter.

```
(define-binary-type id3-frames (tag-size frame-type)
  (:reader (in)
    (loop with to-read = tag-size
          while (plusp to-read)
          for frame = (read-frame frame-type in)
          while frame
          do (decf to-read (+ (frame-header-size frame) (size frame)))
          collect frame
          finally (loop repeat (1- to-read) do (read-byte in))))
  (:writer (out frames)
    (loop with to-write = tag-size
          for frame in frames
          do (write-value frame-type out frame)
          (decf to-write (+ (frame-header-size frame) (size frame)))
          finally (loop repeat to-write do (write-byte 0 out)))))

(defun read-frame (frame-type in)
  (handler-case (read-value frame-type in)
    (in-padding () nil)))
```

The changes are in the calls to `read-frame` and `write-value`, where you need to pass the `frame-type` argument and, in computing the size of the frame, where you need to use a function `frame-header-size` instead of the literal value `6` since the frame header changed size between version 2.2 and version 2.3. Since the

difference in the result of this function is based on the class of the frame, it makes sense to define it as a generic function like this:

```
(defgeneric frame-header-size (frame))
```

You'll define the necessary methods on that generic function in the next section after you define the new frame classes.

Versioned Frame Base Classes

Where before you defined a single base class for all frames, you'll now have two classes, `id3v2.2-frame` and `id3v2.3-frame`. The `id3v2.2-frame` class will be essentially the same as the original `id3-frame` class.

```
(define-tagged-binary-class id3v2.2-frame ()
  ((id (frame-id :length 3))
   (size u3))
  (:dispatch (find-frame-class id)))
```

The `id3v2.3-frame`, on the other hand, requires more changes. The frame identifier and size fields were extended in version 2.3 from three to four bytes each, and two bytes worth of flags were added. Additionally, the frame, like the version 2.3 tag, can contain optional fields, controlled by the values of three of the frame's flags.[8] With those changes in mind, you can define the version 2.3 frame base class, along with some helper functions, like this:

```
(define-tagged-binary-class id3v2.3-frame ()
  ((id                (frame-id :length 4))
   (size              u4)
   (flags             u2)
   (decompressed-size (optional :type 'u4 :if (frame-compressed-p flags)))
   (encryption-scheme (optional :type 'u1 :if (frame-encrypted-p flags)))
   (grouping-identity (optional :type 'u1 :if (frame-grouped-p flags))))
  (:dispatch (find-frame-class id)))

(defun frame-compressed-p (flags) (logbitp 7 flags))

(defun frame-encrypted-p (flags) (logbitp 6 flags))

(defun frame-grouped-p (flags) (logbitp 5 flags))
```

With these two classes defined, you can now implement the methods on the generic function `frame-header-size`.

```
(defmethod frame-header-size ((frame id3v2.2-frame)) 6)

(defmethod frame-header-size ((frame id3v2.3-frame)) 10)
```

The optional fields in a version 2.3 frame aren't counted as part of the header for this computation since they're already included in the value of the frame's `size`.

Versioned Concrete Frame Classes

In the original definition, `generic-frame` subclassed `id3-frame`. But now `id3-frame` has been replaced with the two version-specific base classes, `id3v2.2-frame` and `id3v2.3-frame`. So, you need to define two new versions of `generic-frame`, one for each base class. One way to define this classes would be like this:

```
(define-binary-class generic-frame-v2.2 (id3v2.2-frame)
  ((data (raw-bytes :size size))))

(define-binary-class generic-frame-v2.3 (id3v2.3-frame)
  ((data (raw-bytes :size size))))
```

However, it's a bit annoying that these two classes are the same except for their superclass. It's not too bad in this case since there's only one additional field. But if you take this approach for other concrete frame classes, ones that have a more complex internal structure that's identical between the two ID3 versions, the duplication will be more irksome.

Another approach, and the one you should actually use, is to define a class `generic-frame` as a *mixin*: a class intended to be used as a superclass along with one of the version-specific base classes to produce a concrete, version-specific frame class. The only tricky bit about this approach is that if `generic-frame` doesn't extend either of the frame base classes, then you can't refer to the `size` slot in its definition. Instead, you must use the `current-binary-object` function I discussed at the end of the previous chapter to access the object you're in the midst of reading or writing and pass it to `size`. And you need to account for the difference in the number of bytes of the total frame size that will be left over, in the case of a version 2.3 frame, if

any of the optional fields are included in the frame. So, you should define a generic function `data-bytes` with methods that do the right thing for both version 2.2 and version 2.3 frames.

```
(define-binary-class generic-frame ()
  ((data (raw-bytes :size (data-bytes (current-binary-object))))))

(defgeneric data-bytes (frame))

(defmethod data-bytes ((frame id3v2.2-frame))
  (size frame))

(defmethod data-bytes ((frame id3v2.3-frame))
  (let ((flags (flags frame)))
    (- (size frame)
       (if (frame-compressed-p flags) 4 0)
       (if (frame-encrypted-p flags) 1 0)
       (if (frame-grouped-p flags) 1 0))))
```

Then you can define concrete classes that extend one of the version-specific base classes and `generic-frame` to define version-specific generic frame classes.

```
(define-binary-class generic-frame-v2.2 (id3v2.2-frame generic-frame) ())

(define-binary-class generic-frame-v2.3 (id3v2.3-frame generic-frame) ())
```

With these classes defined, you can redefine the `find-frame-class` function to return the right versioned class based on the length of the identifier.

```
(defun find-frame-class (id)
  (ecase (length id)
    (3 'generic-frame-v2.2)
    (4 'generic-frame-v2.3)))
```

What Frames Do You Actually Need?

With the ability to read both version 2.2 and version 2.3 tags using generic frames, you're ready to start implementing classes to represent the specific frames you care about. However, before you dive in, you should take a breather and figure out what frames you actually care about since, as I mentioned earlier, the ID3 spec specifies many frames that are almost never used. Of course, what frames you care about depends on what kinds of applications you're interested in writing. If you're mostly interested in extracting information from existing ID3 tags, then you need implement only the classes representing the frames containing the information you care about. On the other hand, if you want to write an ID3 tag editor, you may need to support all the frames.

Rather than guessing which frames will be most useful, you can use the code you've already written to poke around a bit at the REPL and see what frames are actually used in your own MP3s. To start, you need an instance of `id3-tag`, which you can get with the `read-id3` function.

```
ID3V2> (read-id3 "/usr2/mp3/Kitka/Wintersongs/02 Byla Cesta.mp3")
#<ID3V2.2-TAG @ #x727b2912>
```

Since you'll want to play with this object a bit, you should save it in a variable.

```
ID3V2> (defparameter *id3* (read-id3 "/usr2/mp3/Kitka/Wintersongs/02 Byla Cesta.mp3"))
*ID3*
```

Now you can see, for example, how many frames it has.

```
ID3V2> (length (frames *id3*))
11
```

Not too many--let's take a look at what they are.

```
ID3V2> (frames *id3*)
(#<GENERIC-FRAME-V2.2 @ #x72dabdda> #<GENERIC-FRAME-V2.2 @ #x72dabec2>
 #<GENERIC-FRAME-V2.2 @ #x72dabfa2> #<GENERIC-FRAME-V2.2 @ #x72dac08a>
 #<GENERIC-FRAME-V2.2 @ #x72dac16a> #<GENERIC-FRAME-V2.2 @ #x72dac24a>
 #<GENERIC-FRAME-V2.2 @ #x72dac32a> #<GENERIC-FRAME-V2.2 @ #x72dac40a>
 #<GENERIC-FRAME-V2.2 @ #x72dac4f2> #<GENERIC-FRAME-V2.2 @ #x72dac632>
 #<GENERIC-FRAME-V2.2 @ #x72dac7b2>)
```

Okay, that's not too informative. What you really want to know are what kinds of frames are in there. In other words, you want to know the `id`s of those frames, which you can get with a simple **MAPCAR** like this:

```
ID3V2> (mapcar #'id (frames *id3*))
("TT2" "TP1" "TAL" "TRK" "TPA" "TYE" "TCO" "TEN" "COM" "COM" "COM")
```

If you look up these identifiers in the ID3v2.2 spec, you'll discover that all the frames with identifiers starting with *T* are text information frames and have a similar structure. And *COM* is the identifier for comment frames, which have a structure similar to that of text information frames. The particular text information frames identified here turn out to be the frames for representing the song title, artist, album, track, part of set, year, genre, and encoding program.

Of course, this is just one MP3 file. Maybe other frames are used in other files. It's easy enough to discover. First define a function that combines the previous **MAPCAR** expression with a call to `read-id3` and wraps the whole thing in a **DELETE-DUPLICATES** to keep things tidy. You'll have to use a `:test` argument of `#'string=` to **DELETE-DUPLICATES** to specify that you want two elements considered the same if they're the same string.

```
(defun frame-types (file)
  (delete-duplicates (mapcar #'id (frames (read-id3 file))) :test #'string=))
```

This should give the same answer except with only one of each identifier when passed the same filename.

```
ID3V2> (frame-types "/usr2/mp3/Kitka/Wintersongs/02 Byla Cesta.mp3")
("TT2" "TP1" "TAL" "TRK" "TPA" "TYE" "TCO" "TEN" "COM")
```

Then you can use Chapter 15's `walk-directory` function along with `mp3-p` to find every MP3 file under a directory and combine the results of calling `frame-types` on each file. Recall that **NUNION** is the recycling version of the **UNION** function; since `frame-types` makes a new list for each file, this is safe.

```
(defun frame-types-in-dir (dir)
  (let ((ids ()))
    (flet ((collect (file)
             (setf ids (nunion ids (frame-types file) :test #'string=))))
      (walk-directory dir #'collect :test #'mp3-p))
    ids))
```

Now pass it the name of a directory, and it'll tell you the set of identifiers used in all the MP3 files under that directory. It may take a few seconds depending how many MP3 files you have, but you'll probably get something similar to this:

```
ID3V2> (frame-types-in-dir "/usr2/mp3/")
("TCON" "COMM" "TRCK" "TIT2" "TPE1" "TALB" "TCP" "TT2" "TP1" "TCM"
 "TAL" "TRK" "TPA" "TYE" "TCO" "TEN" "COM")
```

The four-letter identifiers are the version 2.3 equivalents of the version 2.2 identifiers I discussed previously. Since the information stored in those frames is exactly the information you'll need in Chapter 27, it makes sense to implement classes only for the frames actually used, namely, text information and comment frames, which you'll do in the next two sections. If you decide later that you want to support other frame types, it's mostly a matter of translating the ID3 specifications into the appropriate binary class definitions.

Text Information Frames

All text information frames consist of two fields: a single byte indicating which string encoding is used in the frame and a string encoded in the remaining bytes of the frame. If the encoding byte is zero, the string is encoded in ISO 8859-1; if the encoding is one, the string is a UCS-2 string.

You've already defined binary types representing the four different kinds of strings--two different encodings each with two different methods of delimiting the string. However, `define-binary-class` provides no direct facility for determining the type of value to read based on other values in the object. Instead, you can define a binary type that you pass the value of the encoding byte and that then reads or writes the appropriate kind of string.

As long as you're defining such a type, you can also define it to take two parameters, `:length` and `:terminator`, and pick the right type of string based on which argument is supplied. To implement this new type, you must first define some helper functions. The first two return the name of the appropriate string type based on the encoding byte.

```
(defun non-terminated-type (encoding)
  (ecase encoding
    (0 'iso-8859-1-string)
    (1 'ucs-2-string)))

(defun terminated-type (encoding)
  (ecase encoding
    (0 'iso-8859-1-terminated-string)
    (1 'ucs-2-terminated-string)))
```

Then `string-args` uses the encoding byte, the length, and the terminator to determine several of the arguments to be passed to

`read-value` and `write-value` by the `:reader` and `:writer` of `id3-encoded-string`. One of the length and terminator arguments to `string-args` should always be **`NIL`**.

```
(defun string-args (encoding length terminator)
  (cond
    (length
     (values (non-terminated-type encoding) :length length))
    (terminator
     (values (terminated-type encoding) :terminator terminator))))
```

With those helpers, the definition of `id3-encoded-string` is simple. One detail to note is that the keyword--either `:length` or `:terminator`--used in the call to `read-value` and `write-value` is just another piece of data returned by `string-args`. Although keywords in arguments lists are almost always literal keywords, they don't have to be.

```
(define-binary-type id3-encoded-string (encoding length terminator)
  (:reader (in)
    (multiple-value-bind (type keyword arg)
        (string-args encoding length terminator)
      (read-value type in keyword arg)))
  (:writer (out string)
    (multiple-value-bind (type keyword arg)
        (string-args encoding length terminator)
      (write-value type out string keyword arg))))
```

Now you can define a `text-info` mixin class, much the way you defined `generic-frame` earlier.

```
(define-binary-class text-info-frame ()
  ((encoding u1)
   (information (id3-encoded-string :encoding encoding :length (bytes-left 1)))))
```

As when you defined `generic-frame`, you need access to the size of the frame, in this case to compute the `:length` argument to pass to `id3-encoded-string`. Because you'll need to do a similar computation in the next class you define, you can go ahead and define a helper function, `bytes-left`, that uses `current-binary-object` to get at the size of the frame.

```
(defun bytes-left (bytes-read)
  (- (size (current-binary-object)) bytes-read))
```

Now, as you did with the `generic-frame` mixin, you can define two version-specific concrete classes with a minimum of duplicated code.

```
(define-binary-class text-info-frame-v2.2 (id3v2.2-frame text-info-frame) ())

(define-binary-class text-info-frame-v2.3 (id3v2.3-frame text-info-frame) ())
```

To wire these classes in, you need to modify `find-frame-class` to return the appropriate class name when the ID indicates the frame is a text information frame, namely, whenever the ID starts with *T* and isn't *TXX* or *TXXX*.

```
(defun find-frame-class (name)
  (cond
    ((and (char= (char name 0) #\T)
          (not (member name '("TXX" "TXXX") :test #'string=)))
     (ecase (length name)
       (3 'text-info-frame-v2.2)
       (4 'text-info-frame-v2.3)))
    (t
     (ecase (length name)
       (3 'generic-frame-v2.2)
       (4 'generic-frame-v2.3)))))
```

Comment Frames

Another commonly used frame type is the comment frame, which is like a text information frame with a few extra fields. Like a text information frame, it starts with a single byte indicating the string encoding used in the frame. That byte is followed by a three-character ISO 8859-1 string (regardless of the value of the string encoding byte), which indicates what language the comment is in using an ISO-639-2 code, for example, "eng" for English or "jpn" for Japanese. That field is followed by two strings encoded as indicated by the first byte. The first is a null-terminated string containing a description of the comment. The second, which takes up the remainder of the frame, is the comment text itself.

```
(define-binary-class comment-frame ()
  ((encoding u1)
   (language (iso-8859-1-string :length 3))
   (description (id3-encoded-string :encoding encoding :terminator +null+))
   (text (id3-encoded-string
          :encoding encoding
          :length (bytes-left
                   (+ 1 ; encoding
                      3 ; language
                      (encoded-string-length description encoding t)))))))
```

As in the definition of the `text-info` mixin, you can use `bytes-left` to compute the size of the final string. However, since the `description` field is a variable-length string, the number of bytes read prior to the start of `text` isn't a constant. To make matters worse, the number of bytes used to encode `description` is dependent on the encoding. So, you should define a helper function that returns the number of bytes used to encode a string given the string, the encoding code, and a boolean indicating whether the string is terminated with an extra character.

```
(defun encoded-string-length (string encoding terminated)
  (let ((characters (+ (length string) (if terminated 1 0))))
    (* characters (ecase encoding (0 1) (1 2)))))
```

And, as before, you can define the concrete version-specific comment frame classes and wire them into `find-frame-class`.

```
(define-binary-class comment-frame-v2.2 (id3v2.2-frame comment-frame) ())

(define-binary-class comment-frame-v2.3 (id3v2.3-frame comment-frame) ())

(defun find-frame-class (name)
  (cond
    ((and (char= (char name 0) #\T)
          (not (member name '("TXX" "TXXX") :test #'string=)))
     (ecase (length name)
       (3 'text-info-frame-v2.2)
       (4 'text-info-frame-v2.3)))
    ((string= name "COM")  'comment-frame-v2.2)
    ((string= name "COMM") 'comment-frame-v2.3)
    (t
     (ecase (length name)
       (3 'generic-frame-v2.2)
       (4 'generic-frame-v2.3)))))
```

Extracting Information from an ID3 Tag

Now that you have the basic ability to read and write ID3 tags, you have a lot of directions you could take this code. If you want to develop a complete ID3 tag editor, you'll need to implement specific classes for all the frame types. You'd also need to define methods for manipulating the tag and frame objects in a consistent way (for instance, if you change the value of a string in a `text-info-frame`, you'll likely need to adjust the size); as the code stands, there's nothing to make sure that happens.[9]

Or, if you just need to extract certain pieces of information about an MP3 file from its ID3 tag--as you will when you develop a streaming MP3 server in Chapters 27, 28, and 29--you'll need to write functions that find the appropriate frames and extract the information you want.

Finally, to make this production-quality code, you'd have to pore over the ID3 specs and deal with the details I skipped over in the interest of space. In particular, some of the flags in both the tag and the frame can affect the way the contents of the tag or frame is read; unless you write some code that does the right thing when those flags are set, there may be ID3 tags that this code won't be able to parse correctly. But the code from this chapter should be capable of parsing nearly all the MP3s you actually encounter.

For now you can finish with a few functions to extract individual pieces of information from an `id3-tag`. You'll need these functions in Chapter 27 and probably in other code that uses this library. They belong in this library because they depend on details of the ID3 format that the users of this library shouldn't have to worry about.

To get, say, the name of the song of the MP3 from which an `id3-tag` was extracted, you need to find the ID3 frame with a specific identifier and then extract the information field. And some pieces of information, such as the genre, can require further decoding. Luckily, all the frames that contain the information you'll care about are text information frames, so extracting a particular piece of information mostly boils down to using the right identifier to look up the appropriate frame. Of course, the ID3 authors decided to change all the identifiers between ID3v2.2 and ID3v2.3, so you'll have to account for that.

Nothing too complex--you just need to figure out the right path to get to the various pieces of information. This is a perfect bit of code to develop interactively, much the way you figured out what frame classes you needed to implement. To start, you need an `id3-tag` object to play with. Assuming you have an MP3 laying around, you can use `read-id3` like this:

```
ID3V2> (defparameter *id3* (read-id3 "Kitka/Wintersongs/02 Byla Cesta.mp3"))
*ID3*
ID3V2> *id3*
#<ID3V2.2-TAG @ #x73d04c1a>
```

replacing `Kitka/Wintersongs/02 Byla Cesta.mp3` with the filename of your MP3. Once you have your `id3-tag` object, you can start poking around. For instance, you can check out the list of frame objects with the `frames` function.

```
ID3V2> (frames *id3*)
(#<TEXT-INFO-FRAME-V2.2 @ #x73d04cca>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d04dba>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d04ea2>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d04f9a>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d05082>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d0516a>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d05252>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d0533a>
 #<COMMENT-FRAME-V2.2 @ #x73d0543a>
 #<COMMENT-FRAME-V2.2 @ #x73d05612>
 #<COMMENT-FRAME-V2.2 @ #x73d0586a>)
```

Now suppose you want to extract the song title. It's probably in one of those frames, but to find it, you need to find the frame with the "TT2" identifier. Well, you can check easily enough to see if the tag contains such a frame by extracting all the identifiers like this:

```
ID3V2> (mapcar #'id (frames *id3*))
("TT2" "TP1" "TAL" "TRK" "TPA" "TYE" "TCO" "TEN" "COM" "COM" "COM")
```

There it is, the first frame. However, there's no guarantee it'll always be the first frame, so you should probably look it up by identifier rather than position. That's also straightforward using the **FIND** function.

```
ID3V2> (find "TT2" (frames *id3*) :test #'string= :key #'id)
#<TEXT-INFO-FRAME-V2.2 @ #x73d04cca>
```

Now, to get at the actual information in the frame, do this:

```
ID3V2> (information (find "TT2" (frames *id3*) :test #'string= :key #'id))
"Byla Cesta^@"
```

Whoops. That `^@` is how Emacs prints a null character. In a maneuver reminiscent of the kludge that turned ID3v1 into ID3v1.1, the `information` slot of a text information frame, though not officially a null-terminated string, can contain a null, and ID3 readers are supposed to ignore any characters after the null. So, you need a function that takes a string and returns the contents up to the first null character, if any. That's easy enough using the `+null+` constant from the binary data library.

```
(defun upto-null (string)
  (subseq string 0 (position +null+ string)))
```

Now you can get just the title.

```
ID3V2> (upto-null (information (find "TT2" (frames *id3*) :test #'string= :key #'id)))
"Byla Cesta"
```

You could just wrap that code in a function named `song` that takes an `id3-tag` as an argument, and you'd be done. However, the only difference between this code and the code you'll use to extract the other pieces of information you'll need (such as the album name, the artist, and the genre) is the identifier. So, it's better to split up the code a bit. For starters, you can write a function that just finds a frame given an `id3-tag` and an identifier like this:

```
(defun find-frame (id3 id)
  (find id (frames id3) :test #'string= :key #'id))

ID3V2> (find-frame *id3* "TT2")
#<TEXT-INFO-FRAME-V2.2 @ #x73d04cca>
```

Then the other bit of code, the part that extracts the information from a `text-info-frame`, can go in another function.

```
(defun get-text-info (id3 id)
  (let ((frame (find-frame id3 id)))
    (when frame (upto-null (information frame)))))

ID3V2> (get-text-info *id3* "TT2")
"Byla Cesta"
```

Now the definition of `song` is just a matter of passing the right identifier.

```
(defun song (id3) (get-text-info id3 "TT2"))

ID3V2> (song *id3*)
"Byla Cesta"
```

However, this definition of `song` works only with version 2.2 tags since the identifier changed from "TT2" to "TIT2" between version 2.2 and version 2.3. And all the other tags changed too. Since the user of this library shouldn't have to know about different versions of the ID3 format to do something as simple as get the song title, you should probably handle those details for them. A simple way is to change `find-frame` to take not just a single identifier but a list of identifiers like this:

```
(defun find-frame (id3 ids)
  (find-if #'(lambda (x) (find (id x) ids :test #'string=)) (frames id3)))
```

Then change `get-text-info` slightly so it can take one or more identifiers using a **`&rest`** parameter.

```
(defun get-text-info (id3 &rest ids)
  (let ((frame (find-frame id3 ids)))
    (when frame (upto-null (information frame)))))
```

Then the change needed to allow `song` to support both version 2.2 and version 2.3 tags is just a matter of adding the version 2.3 identifier.

```
(defun song (id3) (get-text-info id3 "TT2" "TIT2"))
```

Then you just need to look up the appropriate version 2.2 and version 2.3 frame identifiers for any fields for which you want to provide an accessor function. Here are the ones you'll need in Chapter 27:

```
(defun album (id3) (get-text-info id3 "TAL" "TALB"))

(defun artist (id3) (get-text-info id3 "TP1" "TPE1"))

(defun track (id3) (get-text-info id3 "TRK" "TRCK"))

(defun year (id3) (get-text-info id3 "TYE" "TYER" "TDRC"))

(defun genre (id3) (get-text-info id3 "TCO" "TCON"))
```

The last wrinkle is that the way the `genre` is stored in the TCO or TCON frames isn't always human readable. Recall that in ID3v1, genres were stored as a single byte that encoded a particular genre from a fixed list. Unfortunately, those codes live on in ID3v2--if the text of the genre frame is a number in parentheses, the number is supposed to be interpreted as an ID3v1 genre code. But, again, users of this library probably won't care about that ancient history. So, you should provide a function that automatically translates the genre. The following function uses the `genre` function just defined to extract the actual genre text and then checks whether it starts with a left parenthesis, decoding the version 1 genre code with a function you'll define in a moment if it does:

```
(defun translated-genre (id3)
  (let ((genre (genre id3)))
```

```
    (if (and genre (char= #\( (char genre 0)))
      (translate-v1-genre genre)
      genre)))
```

Since a version 1 genre code is effectively just an index into an array of standard names, the easiest way to implement `translate-v1-genre` is to extract the number from the genre string and use it as an index into an actual array.

```
(defun translate-v1-genre (genre)
  (aref *id3-v1-genres* (parse-integer genre :start 1 :junk-allowed t)))
```

Then all you need to do is to define the array of names. The following array of names includes the 80 official version 1 genres plus the genres created by the authors of Winamp:

```
(defparameter *id3-v1-genres*
  #(
    ;; These are the official ID3v1 genres.
    "Blues" "Classic Rock" "Country" "Dance" "Disco" "Funk" "Grunge"
    "Hip-Hop" "Jazz" "Metal" "New Age" "Oldies" "Other" "Pop" "R&B" "Rap"
    "Reggae" "Rock" "Techno" "Industrial" "Alternative" "Ska"
    "Death Metal" "Pranks" "Soundtrack" "Euro-Techno" "Ambient"
    "Trip-Hop" "Vocal" "Jazz+Funk" "Fusion" "Trance" "Classical"
    "Instrumental" "Acid" "House" "Game" "Sound Clip" "Gospel" "Noise"
    "AlternRock" "Bass" "Soul" "Punk" "Space" "Meditative"
    "Instrumental Pop" "Instrumental Rock" "Ethnic" "Gothic" "Darkwave"
    "Techno-Industrial" "Electronic" "Pop-Folk" "Eurodance" "Dream"
    "Southern Rock" "Comedy" "Cult" "Gangsta" "Top 40" "Christian Rap"
    "Pop/Funk" "Jungle" "Native American" "Cabaret" "New Wave"
    "Psychadelic" "Rave" "Showtunes" "Trailer" "Lo-Fi" "Tribal"
    "Acid Punk" "Acid Jazz" "Polka" "Retro" "Musical" "Rock & Roll"
    "Hard Rock"

    ;; These were made up by the authors of Winamp but backported into
    ;; the ID3 spec.
    "Folk" "Folk-Rock" "National Folk" "Swing" "Fast Fusion"
    "Bebob" "Latin" "Revival" "Celtic" "Bluegrass" "Avantgarde"
    "Gothic Rock" "Progressive Rock" "Psychedelic Rock" "Symphonic Rock"
    "Slow Rock" "Big Band" "Chorus" "Easy Listening" "Acoustic" "Humour"
    "Speech" "Chanson" "Opera" "Chamber Music" "Sonata" "Symphony"
    "Booty Bass" "Primus" "Porn Groove" "Satire" "Slow Jam" "Club"
    "Tango" "Samba" "Folklore" "Ballad" "Power Ballad" "Rhythmic Soul"
    "Freestyle" "Duet" "Punk Rock" "Drum Solo" "A capella" "Euro-House"
    "Dance Hall"

    ;; These were also invented by the Winamp folks but ignored by the
    ;; ID3 authors.
    "Goa" "Drum & Bass" "Club-House" "Hardcore" "Terror" "Indie"
    "BritPop" "Negerpunk" "Polsk Punk" "Beat" "Christian Gangsta Rap"
    "Heavy Metal" "Black Metal" "Crossover" "Contemporary Christian"
    "Christian Rock" "Merengue" "Salsa" "Thrash Metal" "Anime" "Jpop"
    "Synthpop"))
```

Once again, it probably feels like you wrote a ton of code in this chapter. But if you put it all in a file, or if you download the version from this book's Web site, you'll see it's just not that many lines--most of the pain of writing this library stems from having to understand the intricacies of the ID3 format itself. Anyway, now you have a major piece of what you'll turn into a streaming MP3 server in Chapters 27, 28, and 29. The other major bit of infrastructure you'll need is a way to write server-side Web software, the topic of the next chapter.

---

[1] *Ripping* is the process by which a song on an audio CD is converted to an MP3 file on your hard drive. These days most ripping software also automatically retrieves information about the songs being ripped from online databases such as Gracenote (née the Compact Disc Database [CDDB]) or FreeDB, which it then embeds in the MP3 files as ID3 tags.

[2] Almost all file systems provide the ability to overwrite existing bytes of a file, but few, if any, provide a way to add or remove data at the beginning or middle of a file without having to rewrite the rest of the file. Since ID3 tags are typically stored at the beginning of a file, to rewrite an ID3 tag without disturbing the rest of the file you must replace the old tag with a new tag of exactly the same length. By writing ID3 tags with a certain amount of padding, you have a better chance of being able to do so--if the new tag has more data than the original tag, you use less padding, and if it's shorter, you use more.

[3] The frame data following the ID3 header could also potentially contain the illegal sequence. That's prevented using a different scheme that's turned on via one of the flags in the tag header. The code in this chapter doesn't account for the possibility that this flag might be set; in practice it's rarely used.

[4] In ID3v2.4, UCS-2 is replaced by the virtually identical UTF-16, and UTF-16BE and UTF-8 are added as additional encodings.

[5] The 2.4 version of the ID3 format also supports placing a footer at the end of a tag, which makes it easier to find a tag appended to the end of a file.

[6] Character streams support two functions, `PEEK-CHAR` and `UNREAD-CHAR`, either of which would be a perfect solution to this problem, but binary streams support no equivalent functions.

[7]If a tag had an extended header, you could use this value to determine where the frame data should end. However, if the extended header isn't used, you'd have to use the old algorithm anyway, so it's not worth adding code to do it another way.

[8]These flags, in addition to controlling whether the optional fields are included, can affect the parsing of the rest of the tag. In particular, if the seventh bit of the flags is set, then the actual frame data is compressed using the zlib algorithm, and if the sixth bit is set, the data is encrypted. In practice these options are rarely, if ever, used, so you can get away with ignoring them for now. But that would be an area you'd have to address to make this a production-quality ID3 library. One simple half solution would be to change `find-frame-class` to accept a second argument and pass it the flags; if the frame is compressed or encrypted, you could instantiate a generic frame to hold the data.

[9]Ensuring that kind of interfield consistency would be a fine application for `:after` methods on the accessor generic functions. For instance, you could define this `:after` method to keep `size` in sync with the `information` string:

```
(defmethod (setf information) :after (value (frame text-info-frame))
  (declare (ignore value))
  (with-slots (encoding size information) frame
    (setf size (encoded-string-length information encoding nil))))
```

Copyright Š 2003-2005, Peter Seibel

26. Practical: Web Programming with AllegroServe

In this chapter you'll look at one way to develop Web-based programs in Common Lisp, using the open-source AllegroServe Web server. This isn't meant as a full introduction to AllegroServe. And I'm certainly not going to cover anything more than a tiny corner of the larger topic of Web programming. My goal here is to cover enough of the basics of using AllegroServe that you'll be able, in Chapter 29, to develop an application for browsing a library of MP3 files and streaming them to an MP3 client. Similarly, this chapter will serve as a brief introduction to Web programming for folks new to the topic.

A 30-Second Intro to Server-Side Web Programming

While Web programming today typically involves quite a number of software frameworks and different protocols, the core bits of Web programming haven't changed much since they were invented in the early 1990s. For simple applications, such as the one you'll write in Chapter 29, you need to understand only a few key concepts, so I'll review them quickly here. Experienced Web programmers can skim or skip the rest of this section.[1]

To start, you need to understand the roles the Web browser and the Web server play in Web programming. While a modern browser comes with a lot of bells and whistles, the core functionality of a Web browser is to request Web pages from a Web server and then render them. Typically those pages will be written in the Hypertext Markup Language (HTML), which tells the browser how to render the page, including where to insert inline images and links to other Web pages. HTML consists of text *marked up* with *tags* that give the text a structure that the browser uses when rendering the page. For instance, a simple HTML document looks like this:

```
<html>
  <head>
  <title>Hello</title>
  </head>
  <body>
  <p>Hello, world!</p>
  <p>This is a picture: <img src="some-image.gif"></p>
  <p>This is a <a href="another-page.html">link</a> to another page.</p>
  </body>
</html>
```

Figure 26-1 shows how the browser renders this page.

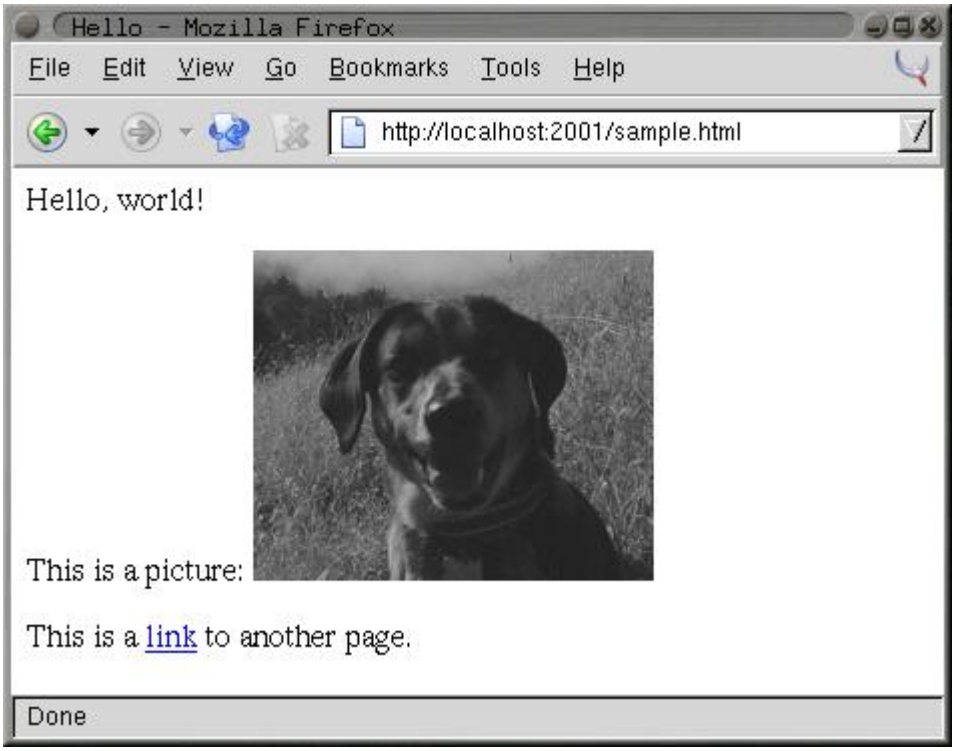


Figure 26-1. Sample Web page

The browser and server communicate using a protocol called the Hypertext Transfer Protocol (HTTP). While you don't need to worry about the details of the protocol, it's worth understanding that it consists entirely of a sequence of requests initiated by the browser and responses generated by the server. That is, the browser connects to the Web server and sends a request that includes, at the least, the desired URL and the version of HTTP that the browser speaks. The browser can also include data in its request; that's how the browser submits HTML forms to the server.

To reply to a request, the server sends a response made up of a set of headers and a body. The headers contain information about the body, such as what type of data it is (for instance, HTML, plain text, or an image), and the body is the data itself, which is then rendered

by the browser. The server can also send an error response telling the browser that its request couldn't be answered for some reason.

And that's pretty much it. Once the browser has received the complete response from the server, there's no communication between the browser and the server until the next time the browser decides to request a page from the server.[2] This is the main constraint of Web programming--there's no way for code running on the server to affect what the user sees in their browser unless the browser issues a new request to the server.[3]

Some Web pages, called *static* pages, are simply HTML files stored on the Web server and served up when requested by the browser. *Dynamic* pages, on the other hand, consist of HTML generated each time the page is requested by a browser. For instance, a dynamic page might be generated by querying a database and then constructing HTML to represent the results of the query.[4]

When generating its response to a request, server-side code has four main pieces of information to act on. The first piece of information is the requested URL. Typically, however, the URL is used by the Web server itself to determine *what* code is responsible for generating the response. Next, if the URL contains a question mark, everything after the question mark is considered to be a *query string*, which is typically ignored by the Web server except that it makes it available to the code generating the response. Most of the time the query string contains a set of key/value pairs. The request from the browser can also contain *post data*, which also usually consists of key/value pairs. Post data is typically used to submit

HTML forms. The key/value pairs supplied in either the query string or the post data are collectively called the *query parameters*.

Finally, in order to string together a sequence of individual requests from the same browser, code running in the server can *set a cookie*, sending a special header in its response to the browser that contains a bit of opaque data called a *cookie*. After a cookie is set by a particular server, the browser will send the cookie with each request it sends to that server. The browser doesn't care about the data in the cookie--it just echoes it back to the server for the server-side code to interpret however it wants.

These are the primitive elements on top of which 99 percent of server-side Web programming is built. The browser sends a request, the server finds some code to handle the request and runs it, and the code uses query parameters and cookies to determine what to do.

AllegroServe

You can serve Web content using Common Lisp in a number of ways; there are at least three open-source Web servers written in Common Lisp as well as plug-ins such as mod_lisp[5] and Lisplets[6] that allow the Apache Web server or any Java Servlet container to delegate requests to a Lisp server running in a separate process.

For this chapter, you'll use a version of the open-source Web server AllegroServe, originally written by John Foderaro at Franz Inc.. AllegroServe is included in the version of Allegro available from Franz for use with this book. If you're not using Allegro, you can use PortableAllegroServe, a friendly fork of the AllegroServe code base, which includes an Allegro compatibility layer that allows PortableAllegroServe to run on most Common Lisps. The code you'll write in this chapter and in Chapter 29 should run in both vanilla AllegroServe and PortableAllegroServe.

AllegroServe provides a programming model similar in spirit to Java Servlets--each time a browser requests a page, AllegroServe parses the request and looks up an object, called an *entity*, which handles the request. Some entity classes provided as part of AllegroServe know how to serve static content--either individual files or the contents of a directory tree. Others, the ones I'll spend most of this chapter discussing, run arbitrary Lisp code to generate the response.[7]

But before I get to that, you need to know how to start AllegroServe and set it up to serve a few files. The first step is to load the

AllegroServe code into your Lisp image. In Allegro, you can simply type `(require :aserve)`. In other Lisps (or in Allegro), you can load PortableAllegroServe by loading the file `INSTALL.lisp` at the top of the `portableaserve` directory tree. Loading AllegroServe will create three new packages, `NET.ASERVE`, `NET.HTML.GENERATOR`, and `NET.ASERVE.CLIENT`.[8]

After loading the server, you start it with the function `start` in the `NET.ASERVE` package. To have easy access to the symbols exported from `NET.ASERVE`, from `COM.GIGAMONKEYS.HTML` (a package I'll discuss in a moment), and from the rest of Common Lisp, you should create a new package to play in like this:

```
CL-USER> (defpackage :com.gigamonkeys.web
           (:use :cl :net.aserve :com.gigamonkeys.html))
#<The COM.GIGAMONKEYS.WEB package>
```

Now switch to that package with this **`IN-PACKAGE`** expression:

```
CL-USER> (in-package :com.gigamonkeys.web)
#<The COM.GIGAMONKEYS.WEB package>
WEB>
```

Now you can use the exported names from `NET.ASERVE` without qualification. The function `start` starts the server. It takes quite a number of keyword parameters, but the only one you need to pass is `:port`, which specifies the port to listen on. You should probably use a high port such as 2001 instead of the default port for HTTP servers, 80, because on Unix-derived operating systems only the root user can listen on ports below 1024. To run AllegroServe listening on port 80 on Unix, you'd need to start Lisp as root and then use the `:setuid` and `:setgid` parameters to tell `start` to switch its

identity after opening the port. You can start a server listening on port 2001 like this:

```
WEB> (start :port 2001)
#<WSERVER port 2001 @ #x72511c72>
```

The server is now running in your Lisp. It's possible you'll get an error that says something about "port already in use" when you try to start the server. This means port 2001 is already in use by some other server on your machine. In that case, the simplest fix is to use a different port, supplying a different argument to `start` and then using that value instead of 2001 in the URLs used throughout this chapter.

You can continue to interact with Lisp via the REPL because AllegroServe starts its own threads to handle requests from browsers. This means, among other things, that you can use the REPL to get a view into the guts of your server while it's running, which makes debugging and testing a lot easier than if the server is a complete black box.

Assuming you're running Lisp on the same machine as your browser, you can check that the server is up and running by pointing your browser at `http://localhost:2001/`. At this point you should get a page-not-found error message in the browser since you haven't published anything yet. But the error message will be from AllegroServe; it'll say so at the bottom of the page. On the other hand, if the browser displays an error dialog that says something like "The connection was refused when attempting to contact localhost:2001," it means either that the server isn't running or that you started it with a different port than 2001.

Now you can publish some files. Suppose you have a file `hello.html` in the directory `/tmp/html` with the following contents:

```
<html>
  <head>
  <title>Hello</title>
  </head>
  <body>
  <p>Hello, world!</p>
  </body>
</html>
```

You can publish it individually with the `publish-file` function.

```
WEB> (publish-file :path "/hello.html" :file "/tmp/html/hello.html")
#<NET.ASERVE::FILE-ENTITY @ #x725eddea>
```

The `:path` argument is the path that will appear in the URL requested by the browser, while the `:file` argument is the name of the file in the file system. After evaluating the `publish-file` expression, you can point your browser to `http://localhost:2001/hello.html`, and it should display a page something like Figure 26-2.

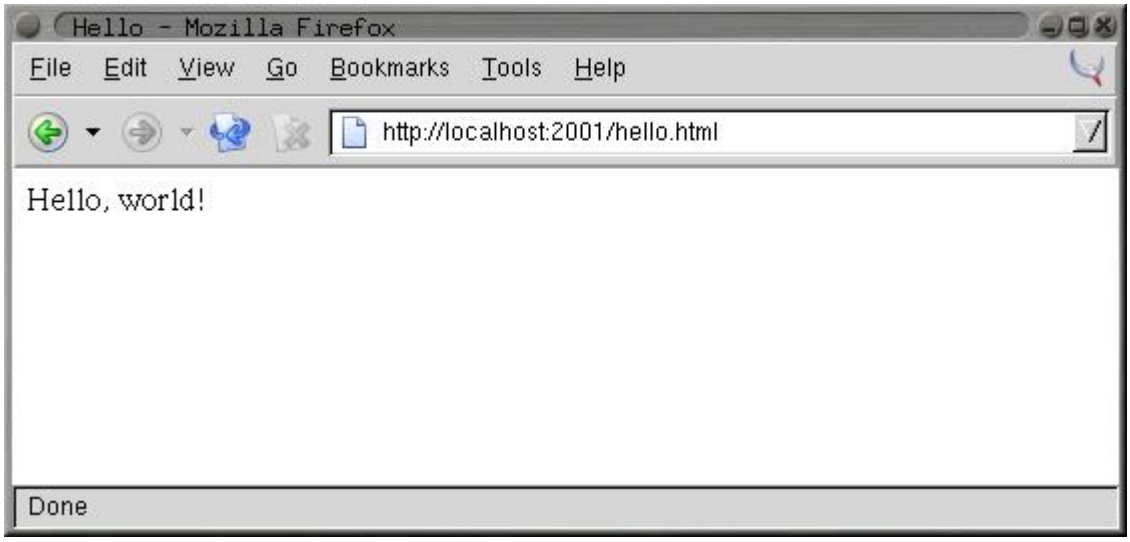


Figure 26-2. *`http://localhost:2001/hello.html`*

You could also publish a whole directory tree of files using the `publish-directory` function. First let's clear out the already published

entity with the following call to `publish-file`:

```
WEB> (publish-file :path "/hello.html" :remove t)
NIL
```

Now you can publish the whole `/tmp/html/` directory (and all its subdirectories) with the `publish-directory` function.

```
WEB> (publish-directory :prefix "/" :destination "/tmp/html/")
#<NET.ASERVE::DIRECTORY-ENTITY @ #x72625aa2>
```

In this case, the `:prefix` argument specifies the beginning of the path part of URLs that should be handled by this entity. Thus, if the server receives a request for `http://localhost:2001/foo/bar.html`, the path is `/foo/bar.html`, which starts with `/`. This path is then translated to a filename by replacing the prefix, `/`, with the destination, `/tmp/html/`. Thus, the URL `http://localhost:2001/hello.html` will still be translated into a request for the file `/tmp/html/hello.html`.

Generating Dynamic Content with AllegroServe

Publishing entities that generate dynamic content is nearly as simple as publishing static content. The functions `publish` and `publish-prefix` are the dynamic analogs of `publish-file` and `publish-directory`. The basic idea of these two functions is that you publish a function that will be called to generate the response to a request for either a specific URL or any URL with a given prefix. The function will be called with two arguments: an object representing the request and the published entity. Most of time you don't need to do anything with the entity object except to pass it along to a couple macros I'll discuss in a moment. On the other hand, you'll use the request object to obtain information submitted by the browser--query parameters included in the URL or data posted using an HTML form.

For a trivial example of using a function to generate dynamic content, let's write a function that generates a page with a different random number each time it's requested.

```
(defun random-number (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (format
       (request-reply-stream request)
       "<html>~@
        <head><title>Random</title></head>~@
        <body>~@
        <p>Random number: ~d</p>~@
        </body>~@
        </html>~@
       "
       (random 1000)))))
```

The macros `with-http-response` and `with-http-body` are part of AllegroServe. The former starts the process of generating an HTTP response and can be used, as here, to specify things such as the

type of content that will be returned. It also handles various parts of HTTP such as dealing with If-Modified-Since requests. The `with-http-body` actually sends the HTTP response headers and then executes its body, which should contain code that generates the content of the reply. Within `with-http-response` but before the `with-http-body`, you can add or change HTTP headers to be sent in the reply. The function `request-reply-stream` is also part of AllegroServe and returns the stream to which you should write output intended to be sent to the browser.

As this function shows, you can just use **`FORMAT`** to print HTML to the stream returned by `request-reply-stream`. In the next section, I'll show you more convenient ways to programmatically generate HTML.[9]

Now you're ready to publish this function.

```
WEB> (publish :path "/random-number" :function 'random-number)
#<COMPUTED-ENTITY @ #x7262bab2>
```

As it does in the `publish-file` function, the `:path` argument specifies the path part of the URL that will result in this function being invoked. The `:function` argument specifies either the name or an actual function object. Using the name of a function, as shown here, allows you to redefine the function later without republishing and have AllegroServe use the new function definition. After evaluating the call to `publish`, you can point your browser at `http:// localhost:2001/random-number` to get a page with a random number on it, as shown in Figure 26-3.

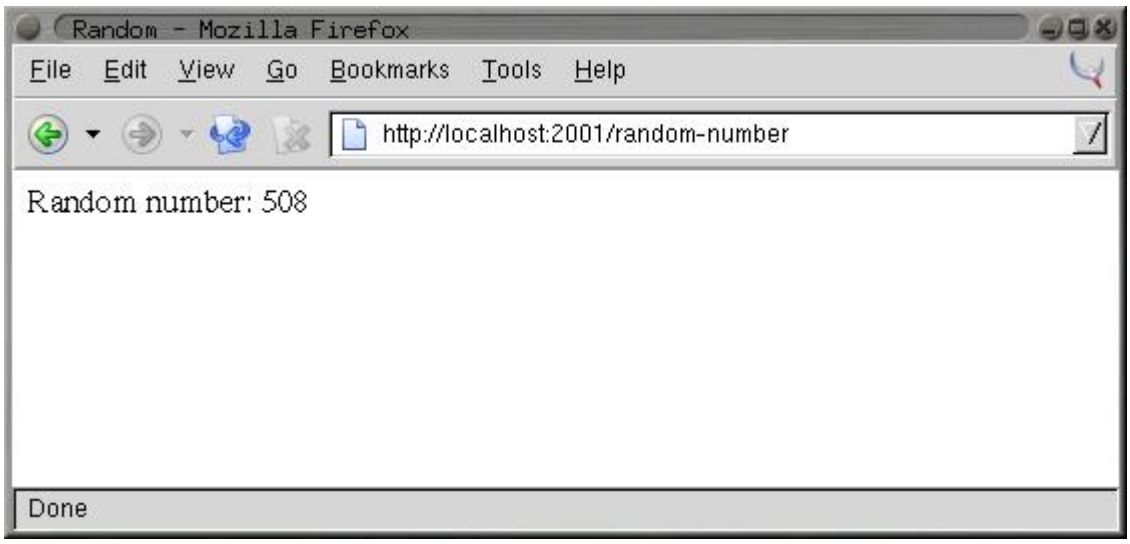


Figure 26-3. `http://localhost:2001/random-number`

Generating HTML

Although using **FORMAT** to emit HTML works fine for the simple pages I've discussed so far, as you start building more elaborate pages it'd be nice to have a more concise way to generate HTML. Several libraries are available for generating HTML from an s-expression representation including one, htmlgen, that's included with AllegroServe. In this chapter you'll use a library called FOO,[10] which is loosely modeled on Franz's htmlgen and whose implementation you'll look at in more detail in Chapters 30 and 31. For now, however, you just need to know how to use FOO.

Generating HTML from within Lisp is quite natural since s-expressions and HTML are essentially isomorphic. You can represent HTML elements with s-expressions by treating each element in HTML as a list "tagged" with an appropriate first element, such as a keyword symbol of the same name as the HTML tag. Thus, the HTML `<p>foo</p>` is represented by the s-expression `(:p "foo")`. Because HTML elements nest the same way lists in s-expressions do, this scheme extends to more complex HTML. For instance, this HTML:

```
<html>
  <head>
  <title>Hello</title>
  </head>
  <body>
  <p>Hello, world!</p>
  </body>
</html>
```

could be represented with the following s-expression:

```
(:html
  (:head (:title "Hello"))
```

```
  (:body (:p "Hello, world!")))
```

HTML elements with attributes complicate things a bit but not in an insurmountable way. FOO supports two ways of including attributes in a tag. One is to simply follow the first item of the list with keyword/value pairs. The first element that follows a keyword/value pair that's not itself a keyword symbol marks the beginning of the element's contents. Thus, you'd represent this HTML:

```
<a href="foo.html">This is a link</a>
```

with the following s-expression:

```
(:a :href "foo.html" "This is a link")
```

The other syntax FOO supports is to group the tag name and attributes into their own list like this:

```
((:a :href "foo.html") "This is link.")
```

FOO can use the s-expression representation of HTML in two ways. The function `emit-html` takes an HTML s-expression and outputs the corresponding HTML.

```
WEB> (emit-html '(:html (:head (:title "Hello")) (:body (:p "Hello, world!"))))
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
T
```

However, `emit-html` isn't always the most efficient way to generate HTML because its argument must be a complete s-expression representation of the HTML to be generated. While it's easy to build such a representation, it's not always particularly efficient. For

instance, suppose you wanted to make an HTML page containing a list of 10,000 random numbers. You could build the s-expression using a backquote template and then pass it to `emit-html` like this:

```
(emit-html
  `(:html
    (:head
      (:title "Random numbers"))
    (:body
      (:h1 "Random numbers")
      (:p ,@(loop repeat 10000 collect (random 1000) collect " ")))))
```

However, this has to build a tree containing a 10,000-element list before it can even start emitting HTML, and the whole s-expression will become garbage as soon as the HTML is emitted. To avoid this inefficiency, FOO also provides a macro `html`, which allows you to embed bits of Lisp code in the middle of an HTML s-expression.

Literal values such as strings and numbers in the input to `html` are interpolated into the output HTML. Likewise, symbols are treated as variable references, and code is generated to emit their value at runtime. Thus, both of these:

```
(html (:p "foo"))

(let ((x "foo")) (html (:p x)))
```

will emit the following:

```
<p>foo</p>
```

List forms that don't start with a keyword symbol are assumed to be code and are embedded in the generated code. Any values the embedded code returns will be ignored, but the code can emit more HTML by calling `html` itself. For instance, to emit the contents of a list in HTML, you might write this:

```
(html (:ul (dolist (item (list 1 2 3)) (html (:li item)))))
```

which will emit the following HTML:

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
```

If you want to emit the value of a list form, you must wrap it in the pseudotag `:print`. Thus, this expression:

```
(html (:p (+ 1 2)))
```

generates this HTML after computing and discarding the value `3`:

```
<p></p>
```

To emit the `3`, you must write this:

```
(html (:p (:print (+ 1 2))))
```

Or you could compute the value and store it in a variable outside the call to `html` like this:

```
(let ((x (+ 1 2))) (html (:p x)))
```

Thus, you can use the `html` macro to generate the list of random numbers like this:

```
(html
  (:html
    (:head
      (:title "Random numbers"))
    (:body
      (:h1 "Random numbers")
      (:p (loop repeat 10 do (html (:print (random 1000)) " "))))))
```

The macro version will be quite a bit more efficient than the `emit-html` version. Not only do you never have to generate an s-expression representing the whole page, also much of the work that

`emit-html` does at runtime to interpret the s-expression will be done once, when the macro is expanded, rather than every time the code is run.

You can control where the output generated by both `html` and `emit-html` is sent with the macro `with-html-output`, which is part of the FOO library. Thus, you can use the `with-html-output` and `html` macros from FOO to rewrite `random-number` like this:

```
(defun random-number (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request))
        (html
          (:html
            (:head (:title "Random"))
            (:body
              (:p "Random number: " (:print (random 1000)))))))))))
```

HTML Macros

Another feature of FOO is that it allows you to define HTML "macros" that can translate arbitrary forms into HTML s-expressions that the `html` macro understands. For instance, suppose you frequently find yourself writing pages of this form:

```
(:html
  (:head (:title "Some title"))
  (:body
    (:h1 "Some title")
    ... stuff ...))
```

You could define an HTML macro to capture that pattern like this:

```
(define-html-macro :standard-page ((&key title) &body body)
  `(:html
     (:head (:title ,title))
     (:body
      (:h1 ,title)
      ,@body)))
```

Now you can use the "tag" `:standard-page` in your s-expression HTML, and it'll be expanded before being interpreted or compiled. For instance, the following:

```
(html (:standard-page (:title "Hello") (:p "Hello, world.")))
```

generates the following HTML:

```
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <h1>Hello</h1>
    <p>Hello, world.</p>
  </body>
</html>
```

Query Parameters

Of course, generating HTML output is only half of Web programming. The other thing you need to do is get input from the user. As I discussed in the "A 30-Second Intro to Server-Side Web Programming" section, when a browser requests a page from a Web server, it can send query parameters in the URL and post data, both of which act as input to the server-side code.

AllegroServe, like most Web programming frameworks, takes care of parsing both these sources of input for you. By the time your published functions are called, all the key/value pairs from the query string and/or post data have been decoded and placed into an alist that you can retrieve from the request object with the function `request-query`. The following function returns a page showing all the query parameters it receives:

```
(defun show-query-params (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request))
        (html
          (:standard-page
           (:title "Query Parameters")
           (if (request-query request)
             (html
               (:table :border 1
                       (loop for (k . v) in (request-query request)
                          do (html (:tr (:td k) (:td v))))))
             (html (:p "No query parameters."))))))))))

(publish :path "/show-query-params" :function 'show-query-params)
```

If you give your browser a URL with a query string in it like the following:

```
http://localhost:2001/show-query-params?foo=bar&baz=10
```

you should get back a page similar to the one shown in Figure 26-4.

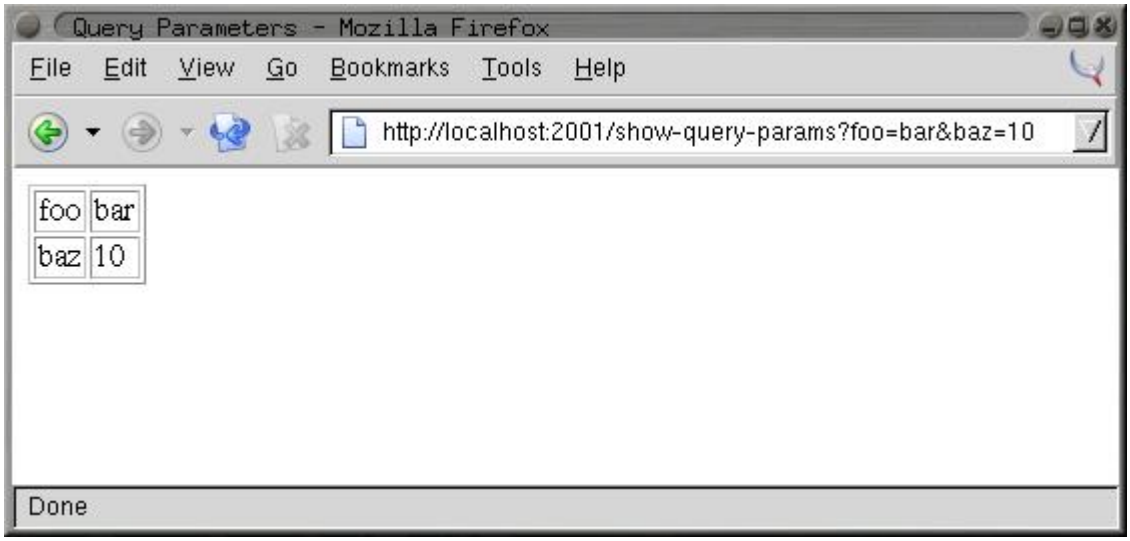


Figure 26-4. *`http://localhost:2001/show-query-params?foo=bar&baz=10`*

To generate some post data, you need an HTML form. The following function generates a simple form, which submits its data to `show-query-params`:

```
(defun simple-form (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (let ((*html-output* (request-reply-stream request)))
        (html
          (:html
            (:head (:title "Simple Form"))
            (:body
             (:form :method "POST" :action "/show-query-params"
               (:table
                (:tr (:td "Foo")
                     (:td (:input :name "foo" :size 20)))
                (:tr (:td "Password")
                     (:td (:input :name "password" :type "password" :size 20))))
               (:p (:input :name "submit" :type "submit" :value "Okay")
                   (:input ::type "reset" :value "Reset")))))))))))

(publish :path "/simple-form" :function 'simple-form)
```

Point your browser to `http://localhost:2001/simple-form`, and you should see a page like the one in Figure 26-5.

If you fill in the form with the "abc" and "def" values, clicking the Okay button should take you to a page like the one in Figure 26-6.

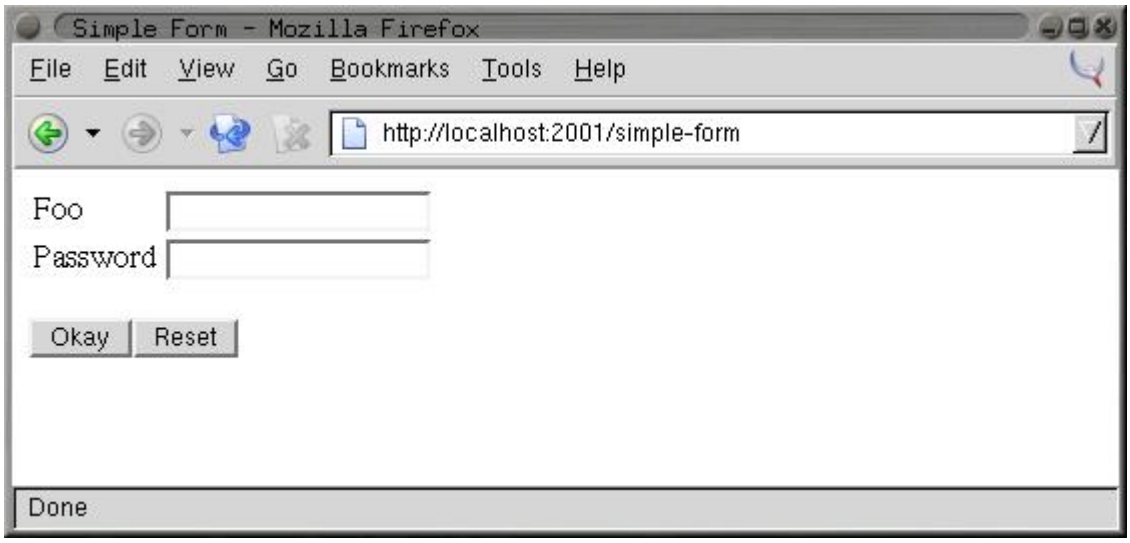


Figure 26-5. `http://localhost:2001/simple-form`

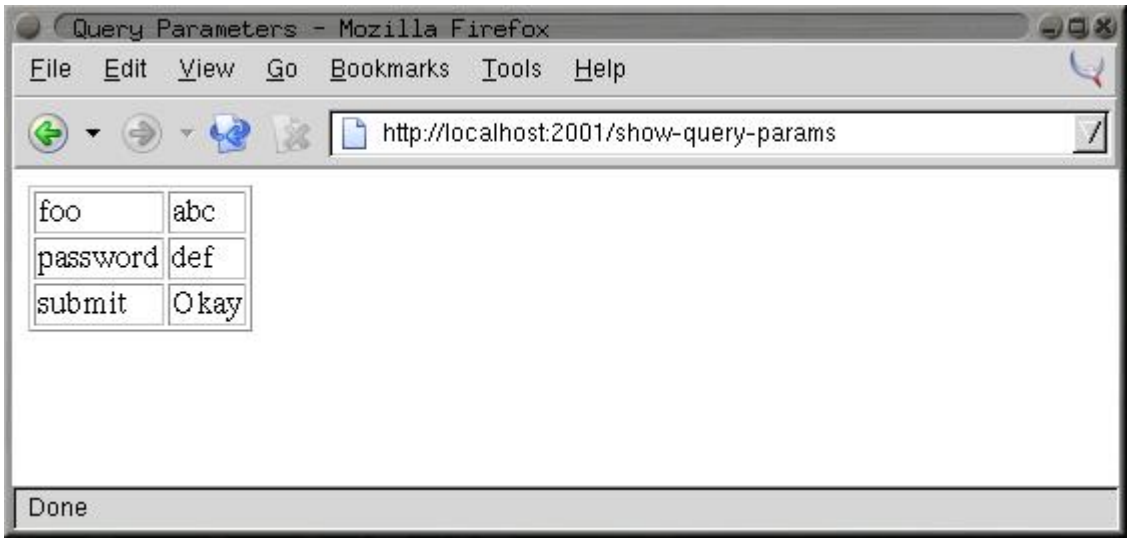


Figure 26-6. Result of submitting the simple form

However, most of the time you won't need to iterate over all the query parameters; you'll want to pick out individual parameters. For instance, you might want to modify `random-number` so the limit value you pass to **RANDOM** can be supplied via a query parameter. In that

case, you use the function `request-query-value`, which takes the request object and the name of the parameter whose value you want and returns the value as a string or **NIL** if no such parameter has been supplied. A parameterizable version of `random-number` might look like this:

```
(defun random-number (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (let* ((*html-output* (request-reply-stream request))
             (limit-string (or (request-query-value "limit" request) ""))
             (limit (or (parse-integer limit-string :junk-allowed t) 1000)))
        (html
          (:html
            (:head (:title "Random"))
            (:body
              (:p "Random number: " (:print (random limit)))))))))
```

Because `request-query-value` can return either **NIL** or an empty string, you have to deal with both those cases when parsing the parameter into a number to pass to **RANDOM**. You can deal with a **NIL** value when you bind `limit-string`, binding it to `""` if there's no "limit" query parameter. Then you can use the `:junk-allowed` argument to **PARSE-INTEGER** to ensure that it returns either **NIL** (if it can't parse an integer from the string given) or an integer. In the section "A Small Application Framework," you'll develop some macros to make it easier to deal with grabbing query parameters and converting them to various types.

Cookies

In AllegroServe you can send a Set-Cookie header that tells the browser to save a cookie and send it along with subsequent requests by calling the function `set-cookie-header` within the body of `with-http-response` but before the call to `with-http-body`. The first argument to the function is the request object, and the remaining arguments are keyword arguments used to set the various properties of the cookie. The only two you must pass are the `:name` and `:value` arguments, both of which should be strings. The other possible arguments that affect the cookie sent to the browser are `:expires`, `:path`, `:domain`, and `:secure`.

Of these, you need to worry only about `:expires`. It controls how long the browser should save the cookie. If `:expires` is **NIL** (the default), the browser will save the cookie only until it exits. Other possible values are `:never`, which means the cookie should be kept forever, or a universal time as returned by **GET-UNIVERSAL-TIME** or **ENCODE-UNIVERSAL-TIME**. An `:expires` of zero tells the client to immediately discard an existing cookie.[11]

After you've set a cookie, you can use the function `get-cookie-values` to get an alist containing one name/value pair for each cookie sent by the browser. From that alist, you can pick out individual cookie values using **ASSOC** and **CDR**.

The following function shows the names and values of all the cookies sent by the browser:

```
(defun show-cookies (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request))
        (html
          (:standard-page
           (:title "Cookies")
           (if (null (get-cookie-values request))
             (html (:p "No cookies."))
             (html
               (:table
                 (loop for (key . value) in (get-cookie-values request)
                    do (html (:tr (:td key) (:td value))))))))))))

(publish :path "/show-cookies" :function 'show-cookies)
```

The first time you load the page `http://localhost:2001/show-cookies` it should say "No cookies" as shown in Figure 26-7 since you haven't set any yet.

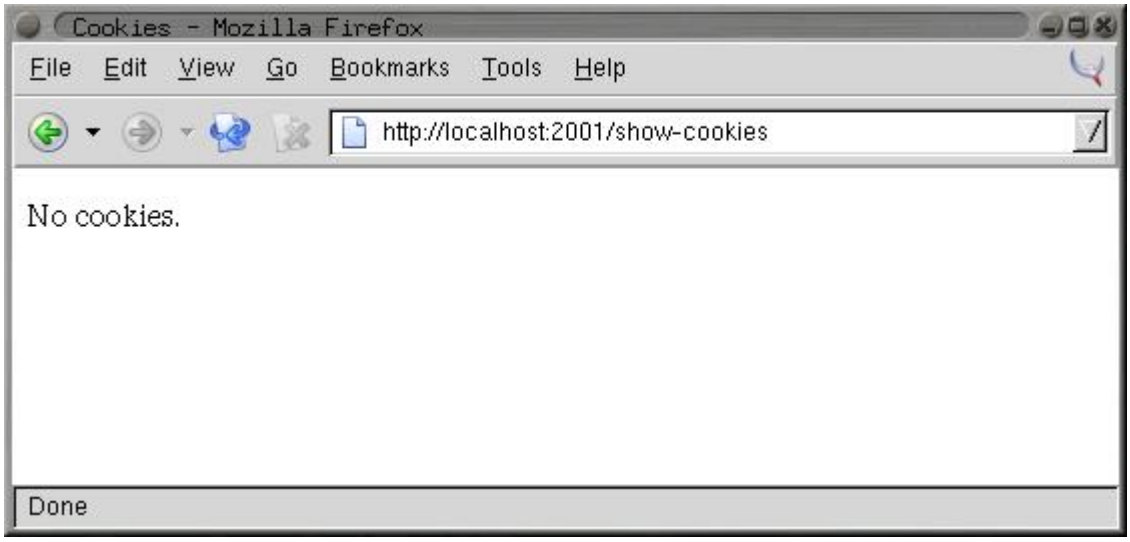


Figure 26-7. `http://localhost:2001/show-cookies` with no cookies

To set a cookie, you need another function, such as the following:

```
(defun set-cookie (request entity)
  (with-http-response (request entity :content-type "text/html")
    (set-cookie-header request :name "MyCookie" :value "A cookie value")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request))
        (html
          (:standard-page
           (:title "Set Cookie")
           (:p "Cookie set.")
           (:p (:a :href "/show-cookies" "Look at cookie jar."))))))))
```

```
(publish :path "/set-cookie" :function 'set-cookie)
```

If you enter the URL `http://localhost:2001/set-cookie`, your browser should display a page like the one in Figure 26-8. Additionally, the server will send a Set-Cookie header with a cookie named "MyCookie" with "A cookie value" as its value. If you click the link *Look at cookie jar*, you'll be taken to the `/show-cookies` page where you'll see the new cookie, as shown in Figure 26-9. Because you didn't specify an `:expires` argument, the browser will continue to send the cookie with each request until you quit the browser.

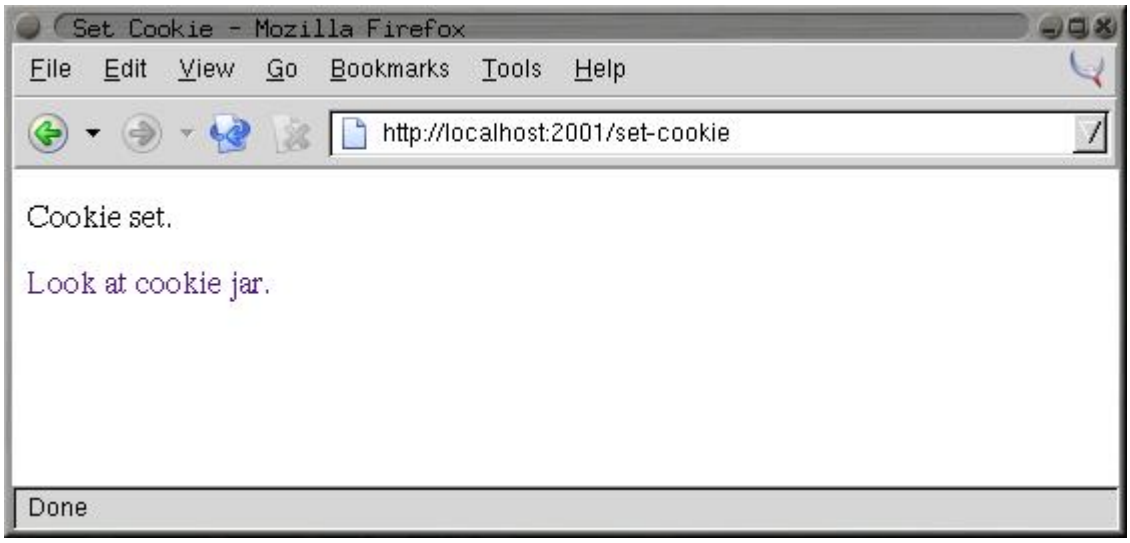


Figure 26-8. *`http://localhost:2001/set-cookie`*

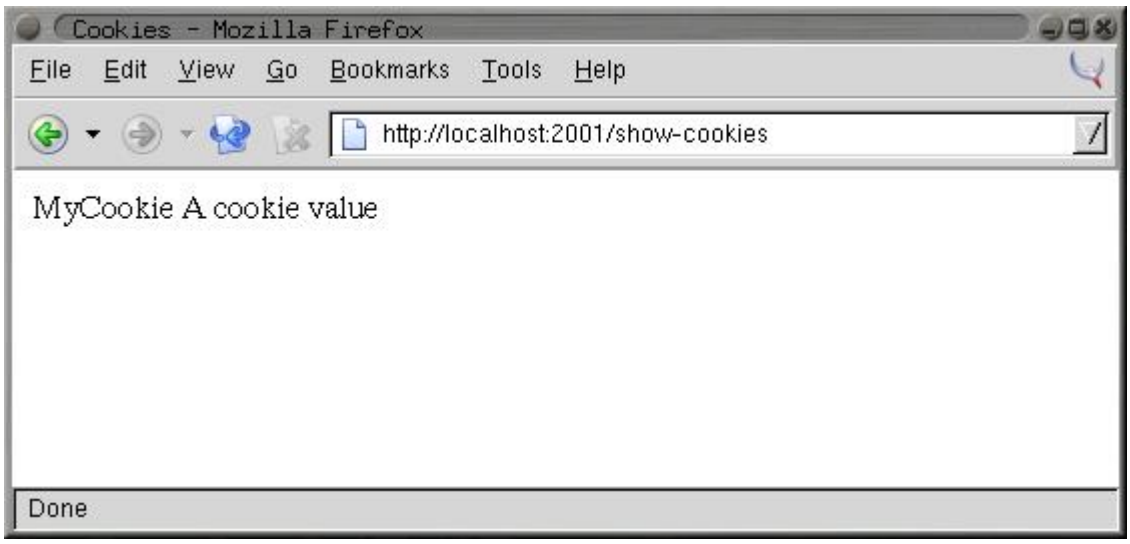


Figure 26-9. `http://localhost:2001/show-cookies` *after setting a cookie*

A Small Application Framework

Although AllegroServe provides fairly straightforward access to all the basic facilities you need to write server-side Web code (access to query parameters from both the URL's query string and the post data; the ability to set cookies and retrieve their values; and, of course, the ability to generate the response sent back to the browser), there's a fair bit of annoyingly repetitive code.

For instance, every HTML-generating function you write is going to take the arguments `request` and `entity` and then will contain calls to `with-http-response`, `with-http-response`, and--if you're going to use FOO to generate HTML--`with-html-output`. Then, in functions that need to get at query parameters, there will be a bunch of calls to `request-query-value` and then more code to convert the string returned to whatever type you actually want. Finally, you need to remember to `publish` the function.

To reduce the amount of boilerplate you have to write, you can write a small framework on top of AllegroServe to make it easier to define functions that handle requests for a particular URL.

The basic approach will be to define a macro, `define-url-function`, that you'll use to define functions that will automatically be published via `publish`. This macro will expand into a **`DEFUN`** that contains the appropriate boilerplate as well as code to publish the function under a URL of the same name. It'll also take care of generating code to extract values from query parameters and cookies and to bind them

to variables declared in the function's parameter list. Thus, the basic form of a `define-url-function` definition is this:

```
(define-url-function name (request query-parameter*)
  body)
```

where the *body* is the code to emit the HTML of the page. It'll be wrapped in a call to FOO's `html` macro, so for simple pages it might contain nothing but s-expression HTML.

Within the body, the query parameter variables will be bound to values of query parameters with the same name or from a cookie. In the simplest case, a query parameter's value will be the string taken from the query parameter or post data field of the same name. If the query parameter is specified with a list, you can also specify an automatic type conversion, a default value, and whether to look for and save the value of the parameter in a cookie. The complete syntax for a *query-parameter* is as follows:

```
name | (name type [default-value] [stickiness])
```

The *type* must be a name recognized by `define-url-function`. I'll discuss in a moment how to define new types. The *default-value* must be a value of the given type. Finally, *stickiness*, if supplied, indicates that the parameter's value should be taken from an appropriately named cookie if no query parameter is supplied and that a Set-Cookie header should be sent in the response that saves the value in the cookie of the same name. Thus, a sticky parameter, after being explicitly supplied a value via a query parameter, will keep that value on subsequent requests of the page even when no query parameter is supplied.

The name of the cookie used depends on the value of *stickiness*: with a value of `:global`, the cookie will be named the same as the parameter. Thus, different functions that use globally sticky parameters with the same name will share the value. If *stickiness* is `:package`, then the cookie name is constructed from the name of the parameter and the package of the function's name; this allows functions in the same package to share values but not have to worry about stomping on parameters of functions in other packages. Finally, a parameter with a *stickiness* value of `:local` will use a cookie made from the name of the parameter, the package of the function name, and the function name, making it unique to that function.

For instance, you can use `define-url-function` to replace the previous eleven-line definition of `random-page` with this five-line version:

```
(define-url-function random-number (request (limit integer 1000))
  (:html
    (:head (:title "Random"))
    (:body
      (:p "Random number: " (:print (random limit))))))
```

If you wanted the limit argument to be sticky, you could change the limit declaration to `(limit integer 1000 :local)`.

The Implementation

I'll explain the implementation of `define-url-function` from the top down. The macro itself looks like this:

```
(defmacro define-url-function (name (request &rest params) &body body)
  (with-gensyms (entity)
    (let ((params (mapcar #'normalize-param params)))
      `(progn
         (defun ,name (,request ,entity)
           (with-http-response (,request ,entity :content-type "text/html")
             (let* (,@(param-bindings name request params))
               ,@(set-cookies-code name request params)
               (with-http-body (,request ,entity)
                 (with-html-output ((request-reply-stream ,request))
                   (html ,@body))))))
         (publish :path ,(format nil "/~(~a~)" name) :function ',name)))))
```

Let's take it bit by bit, starting with the first few lines.

```
(defmacro define-url-function (name (request &rest params) &body body)
  (with-gensyms (entity)
    (let ((params (mapcar #'normalize-param params)))
```

Up to here you're just getting ready to generate code. You **GENSYM** a symbol to use later as the name of the entity parameter in the **DEFUN**. Then you normalize the parameters, converting plain symbols to list form using this function:

```
(defun normalize-param (param)
  (etypecase param
    (list param)
    (symbol `(,param string nil nil))))
```

In other words, declaring a parameter with just a symbol is the same as declaring a nonsticky, string parameter with no default value.

Then comes the **PROGN**. You must expand into a **PROGN** because you need to generate code to do two things: define a function with **DEFUN** and call `publish`. You should define the function first so if there's an error in the definition, the function won't be published. The first two lines of the **DEFUN** are just boilerplate.

```
(defun ,name (,request ,entity)
  (with-http-response (,request ,entity :content-type "text/html")
```

Now you do the real work. The following two lines generate the bindings for the parameters specified in `define-url-function` other than `request` and the code that calls `set-cookie-header` for the sticky parameters. Of course, the real work is done by helper functions that you'll look at in a moment.[12]

```
      (let* (,@(param-bindings name request params))
        ,@(set-cookies-code name request params)
```

The rest is just more boilerplate, putting the body from the `define-url-function` definition in the appropriate context of `with-http-body`, `with-html-output`, and `html` macros. Then comes the call to `publish`.

```
     (publish :path ,(format nil "/~(~a~)" name) :function ',name)
```

The expression `(format nil "/~(~a~)" name)` is evaluated at macro expansion time, generating a string consisting of /, followed by an all-lowercase version of the name of the function you're about to define. That string becomes the `:path` argument to publish, while the function name is interpolated as the `:function` argument.

Now let's look at the helper functions used to generate the **DEFUN** form. To generate parameter bindings, you need to loop over the `params` and collect a snippet of code for each one, generated by `param-binding`. That snippet will be a list containing the name of the variable to bind and the code that will compute the value of that variable. The exact form of code used to compute the value will depend on the type of the parameter, whether it's sticky, and the default value, if any. Because you already normalized the params, you can use **DESTRUCTURING-BIND** to take them apart in `param-binding`.

```
(defun param-bindings (function-name request params)
  (loop for param in params
     collect (param-binding function-name request param)))

(defun param-binding (function-name request param)
  (destructuring-bind (name type &optional default sticky) param
    (let ((query-name (symbol->query-name name))
          (cookie-name (symbol->cookie-name function-name name sticky)))
      `(,name (or
               (string->type ',type (request-query-value ,query-name ,request))
               ,@(if cookie-name
                     (list `(string->type ',type (get-cookie-value ,request ,cookie-name))))
               ,default)))))
```

The function `string->type`, which you use to convert strings obtained from the query parameters and cookies to the desired type, is a generic function with the following signature:

```
(defgeneric string->type (type value))
```

To make a particular name usable as a type name for a query parameter, you just need to define a method on `string->type`. You'll need to define at least a method specialized on the symbol `string` since that's the default type. Of course, that's pretty easy. Since browsers sometimes submit forms with empty strings to indicate no value was supplied for a particular value, you'll want to convert an empty string to `NIL` as this method does:

```
(defmethod string->type ((type (eql 'string)) value)
  (and (plusp (length value)) value))
```

You can add conversions for other types needed by your application. For instance, to make `integer` usable as a query parameter type so you can handle the `limit` parameter of `random-page`, you might define this method:

```
(defmethod string->type ((type (eql 'integer)) value)
  (parse-integer (or value "") :junk-allowed t))
```

Another helper function used in the code generated by `param-binding` is `get-cookie-value`, which is just a bit of sugar around the `get-cookie-values` function provided by AllegroServe. It looks like this:

```
(defun get-cookie-value (request name)
  (cdr (assoc name (get-cookie-values request) :test #'string=)))
```

The functions that compute the query parameter and cookies names are similarly straightforward.

```
(defun symbol->query-name (sym)
  (string-downcase sym))

(defun symbol->cookie-name (function-name sym sticky)
  (let ((package-name (package-name (symbol-package function-name))))
    (when sticky
      (ecase sticky
        (:global
         (string-downcase sym))
        (:package
         (format nil "~(~a:~a~)" package-name sym))
```

```
        (:local
         (format nil "~(~a:~a:~a~)" package-name function-name sym))))))
```

To generate the code that sets cookies for sticky parameters, you again loop over the list of parameters, this time collecting a snippet of code for each sticky param. You can use the `when` and `collect it` **LOOP** forms to collect only the non-**NIL** values returned by `set-cookie-code`.

```
(defun set-cookies-code (function-name request params)
  (loop for param in params
        when (set-cookie-code function-name request param) collect it))

(defun set-cookie-code (function-name request param)
  (destructuring-bind (name type &optional default sticky) param
    (declare (ignore type default))
    (if sticky
      `(when ,name
         (set-cookie-header
          ,request
          :name ,(symbol->cookie-name function-name name sticky)
          :value (princ-to-string ,name))))))
```

One of the advantages of defining macros in terms of helper functions like this is that it's easy to make sure the individual bits of code you're generating look right. For instance, you can check that the following `set-cookie-code`:

```
(set-cookie-code 'foo 'request '(x integer 20 :local))
```

generates something like this:

```
(WHEN X
  (SET-COOKIE-HEADER REQUEST
    :NAME "com.gigamonkeys.web:foo:x"
    :VALUE (PRINC-TO-STRING X)))
```

Assuming this code will occur in a context where `x` is the name of a variable, this looks good.

Once again, macros have allowed you to distill the code you need to write down to its essence--in this case, the data you want to extract from the request and the HTML you want to generate. That said, this framework isn't meant to be the be-all and end-all of Web application frameworks--it's just a little sugar to make it a bit easier to write simple apps like the one you'll write in Chapter 29.

But before you can get to that, you need to write the guts of the application for which the Chapter 29 application will be the user interface. You'll start in the next chapter with a souped-up version of the database you wrote in Chapter 3, this time to keep track of ID3 data extracted from MP3 files.

---

[1]Readers new to Web programming will probably need to supplement this introduction with a more in-depth tutorial or two. You can find a good set of online tutorials at `http://www.jmarshall.com/easy/`.

[2]Loading a single Web page may actually involve multiple requests--to render the HTML of a page containing inline images, the browser must request each image individually and then insert each into the appropriate place in the rendered HTML.

[3]Much of the complexity around Web programming is a result of trying to work around this fundamental limitation in order to provide a user experience that's more like the interactivity provided by desktop applications.

[4]Unfortunately, *dynamic* is somewhat overloaded in the Web world. The phrase *Dynamic HTML* refers to HTML containing embedded code, usually in the language JavaScript, that can be executed in the browser without further communication with the Web server. Used with some discretion, Dynamic HTML can improve the usability of a Web-based application since, even with high-speed Internet connections, making a request to a Web server, receiving the response, and rendering the new page can take a noticeable amount of time. To further confuse things, dynamically generated pages (in other words, generated on the server) could also contain Dynamic HTML (code to be run on the client.) For the purposes of this book, you'll stick to dynamically generating plain old nondynamic HTML.

[5]`http://www.fractalconcept.com/asp/html/mod_lisp.html`

[6]`http://lisplets.sourceforge.net/`

[7]AllegroServe also provides a framework called *Webactions* that's analogous to JSPs in the Java world--instead of writing code that generates HTML, with Webactions you write pages that are essentially HTML with a bit of magic foo that turns into code to be run when the page is served. I won't cover Webactions in this book.

[8]Loading PortableAllegroServe will create some other packages for the compatibility libraries, but the packages you'll care about are those three.

[9]The `~@` followed by a newline tells `FORMAT` to ignore whitespace after the newline, which allows you to indent your code nicely without adding a bunch of whitespace to the HTML. Since white-space is typically not significant in HTML, this doesn't matter to the browser, but it makes the generated HTML source look a bit nicer to humans.

[10]FOO is a recursive tautological acronym for *FOO Outputs Output*.

[11]For information about the meaning of the other parameters, see the AllegroServe documentation and RFC 2109, which describes the cookie mechanism.

[12]You need to use **LET*** rather than a **LET** to allow the default value forms for parameters to refer to parameters that appear earlier in the parameter list. For example, you could write this:

```
(define-url-function (request (x integer 10) (y integer (* 2 x))) ...)
```

and the value of `y`, if not explicitly supplied, would be twice the value of `x`.

Copyright © 2003-2005, Peter Seibel

27. Practical: An MP3 Database

In this chapter you'll revisit the idea first explored in Chapter 3 of building an in-memory database out of basic Lisp data structures. This time your goal is to hold information that you'll extract from a collection of MP3 files using the ID3v2 library from Chapter 25. You'll then use this database in Chapters 28 and 29 as part of a Web-based streaming MP3 server. Of course, this time around you can use some of the language features you've learned since Chapter 3 to build a more sophisticated version.

The Database

The main problem with the database in Chapter 3 is that there's only one table, the list stored in the variable `*db*`. Another is that the code doesn't know anything about what type of values are stored in different columns. In Chapter 3 you got away with that by using the fairly general-purpose **EQUAL** method to compare column values when selecting rows from the database, but you would've been in trouble if you had wanted to store values that couldn't be compared with **EQUAL** or if you had wanted to sort the rows in the database since there's no ordering function that's as general as **EQUAL**.

This time you'll solve both problems by defining a class, `table`, to represent individual database tables. Each `table` instance will consist of two slots--one to hold the table's data and another to hold information about the columns in the table that database operations will be able to use. The class looks like this:

```
(defclass table ()
  ((rows   :accessor rows   :initarg :rows :initform (make-rows))
   (schema :accessor schema :initarg :schema)))
```

As in Chapter 3, you can represent the individual rows with plists, but this time around you'll create an abstraction that will make that an implementation detail you can change later without too much trouble. And this time you'll store the rows in a vector rather than a list since certain operations that you'll want to support, such as random access to rows by a numeric index and the ability to sort a table, can be more efficiently implemented with vectors.

The function `make-rows` used to initialize the `rows` slot can be a simple wrapper around **`MAKE-ARRAY`** that builds an empty, adjustable,vector with a fill pointer.

The Package

The package for the code you'll develop in this chapter looks like this:

```
(defpackage :com.gigamonkeys.mp3-database
  (:use :common-lisp
        :com.gigamonkeys.pathnames
        :com.gigamonkeys.macro-utilities
        :com.gigamonkeys.id3v2)
  (:export  :*default-table-size*
            :*mp3-schema*
            :*mp3s*
            :column
            :column-value
            :delete-all-rows
            :delete-rows
            :do-rows
            :extract-schema
            :in
            :insert-row
            :load-database
            :make-column
            :make-schema
            :map-rows
            :matching
            :not-nullable
            :nth-row
            :random-selection
            :schema
            :select
            :shuffle-table
            :sort-rows
            :table
            :table-size
            :with-column-values))
```

The `:use` section gives you access to the functions and macros whose names are exported from the packages defined in Chapter 15, 8, and 25 and the `:export` section exports the API this library will provide, which you'll use in Chapter 29.

```
(defparameter *default-table-size* 100)

(defun make-rows (&optional (size *default-table-size*))
  (make-array size :adjustable t :fill-pointer 0))
```

To represent a table's schema, you need to define another class, `column`, each instance of which will contain information about one column in the table: its name, how to compare values in the column for equality and ordering, a default value, and a function that will be used to normalize the column's values when inserting data into the table and when querying the table. The `schema` slot will hold a list of `column` objects. The class definition looks like this:

```
(defclass column ()
  ((name
    :reader name
    :initarg :name)

   (equality-predicate
    :reader equality-predicate
    :initarg :equality-predicate)

   (comparator
    :reader comparator
    :initarg :comparator)

   (default-value
    :reader default-value
    :initarg :default-value
    :initform nil)

   (value-normalizer
    :reader value-normalizer
    :initarg :value-normalizer
    :initform #'(lambda (v column) (declare (ignore column)) v))))
```

The `equality-predicate` and `comparator` slots of a `column` object hold functions used to compare values from the given column for equivalence and ordering. Thus, a column containing string values might have **`STRING=`** as its `equality-predicate` and **`STRING<`** as its `comparator`, while a column containing numbers might have `=` and `<`.

The `default-value` and `value-normalizer` slots are used when inserting rows into the database and, in the case of `value-normalizer`, when querying the database. When you insert a row into the database, if no value is provided for a particular column,

you can use the value stored in the `column`'s `default-value` slot. Then the value--defaulted or otherwise--is normalized by passing it and the column object to the function stored in the `value-normalizer` slot. You pass the column in case the `value-normalizer` function needs to use some data associated with the column object. (You'll see an example of this in the next section.) You should also normalize values passed in queries before comparing them with values in the database.

Thus, the `value-normalizer`'s responsibility is primarily to return a value that can be safely and correctly passed to the `equality-predicate` and `comparator` functions. If the `value-normalizer` can't figure out an appropriate value to return, it can signal an error.

The other reason to normalize values before you store them in the database is to save both memory and CPU cycles. For instance, if you have a column that's going to contain string values but the number of distinct strings that will be stored in the column is small--for instance, the genre column in the MP3 database--you can save space and speed by using the `value-normalizer` to *intern* the strings (translate all **`STRING=`** values to a single string object). Thus, you'll need only as many strings as there are distinct values, regardless of how many rows are in the table, and you can use **`EQL`** to compare column values rather than the slower **`STRING=`**.[1]

Defining a Schema

Thus, to make an instance of `table`, you need to build a list of `column` objects. You could build the list by hand, using **`LIST`** and **`MAKE-INSTANCE`**. But you'll soon notice that you're frequently making a lot column objects with the same comparator and equality-predicate combinations. This is because the combination of a comparator and equality predicate essentially defines a column type. It'd be nice if there was a way to give those types names that would allow you to say simply that a given column is a string column, rather than having to specify **`STRING<`** as its comparator and **`STRING=`** as its equality predicate. One way is to define a generic function, `make-column`, like this:

```
(defgeneric make-column (name type &optional default-value))
```

Now you can implement methods on this generic function that specialize on `type` with **`EQL`** specializers and return `column` objects with the slots filled in with appropriate values. Here's the generic function and methods that define column types for the type names `string` and `number`:

```
(defmethod make-column (name (type (eql 'string)) &optional default-value)
  (make-instance
   'column
   :name name
   :comparator #'string<
   :equality-predicate #'string=
   :default-value default-value
   :value-normalizer #'not-nullable))

(defmethod make-column (name (type (eql 'number)) &optional default-value)
  (make-instance
   'column
   :name name
   :comparator #'<
   :equality-predicate #'=
   :default-value default-value))
```

The following function, `not-nullable`, used as the `value-normalizer` for `string` columns, simply returns the value it's given unless the value is **NIL**, in which case it signals an error:

```
(defun not-nullable (value column)
  (or value (error "Column ~a can't be null" (name column))))
```

This is important because **STRING<** and **STRING=** will signal an error if called on **NIL**; it's better to catch bad values before they go into the table rather than when you try to use them.[2]

Another column type you'll need for the MP3 database is an `interned-string` whose values are interned as discussed previously. Since you need a hash table in which to intern values, you should define a subclass of `column`, `interned-values-column`, that adds a slot whose value is the hash table you use to intern.

To implement the actual interning, you'll also need to provide an `:initform` for `value-normalizer` of a function that interns the value in the column's `interned-values` hash table. And because one of the main reasons to intern values is to allow you to use **EQL** as the equality predicate, you should also add an `:initform` for the `equality-predicate` of `#'eql`.

```
(defclass interned-values-column (column)
  ((interned-values
    :reader interned-values
    :initform (make-hash-table :test #'equal))
   (equality-predicate :initform #'eql)
   (value-normalizer   :initform #'intern-for-column)))

(defun intern-for-column (value column)
  (let ((hash (interned-values column)))
    (or (gethash (not-nullable value column) hash)
        (setf (gethash value hash) value))))
```

You can then define a `make-column` method specialized on the name `interned-string` that returns an instance of `interned-values-column`.

```
(defmethod make-column (name (type (eql 'interned-string)) &optional default-value)
  (make-instance
   'interned-values-column
   :name name
   :comparator #'string<
   :default-value default-value))
```

With these methods defined on `make-column`, you can now define a function, `make-schema`, that builds a list of `column` objects from a list of column specifications consisting of a column name, a column type name, and, optionally, a default value.

```
(defun make-schema (spec)
  (mapcar #'(lambda (column-spec) (apply #'make-column column-spec)) spec))
```

For instance, you can define the schema for the table you'll use to store data extracted from MP3s like this:

```
(defparameter *mp3-schema*
  (make-schema
   '((:file     string)
     (:genre    interned-string "Unknown")
     (:artist   interned-string "Unknown")
     (:album    interned-string "Unknown")
     (:song     string)
     (:track    number 0)
     (:year     number 0)
     (:id3-size number))))
```

To make an actual table for holding information about MP3s, you pass `*mp3-schema*` as the `:schema` initarg to **`MAKE-INSTANCE`**.

```
(defparameter *mp3s* (make-instance 'table :schema *mp3-schema*))
```

Inserting Values

Now you're ready to define your first table operation, `insert-row`, which takes a plist of names and values and a table and adds a row to the table containing the given values. The bulk of the work is done in a helper function, `normalize-row`, that builds a plist with a defaulted, normalized value for each column, using the values from `names-and-values` if available and the `default-value` for the column if not.

```
(defun insert-row (names-and-values table)
  (vector-push-extend (normalize-row names-and-values (schema table)) (rows table)))

(defun normalize-row (names-and-values schema)
  (loop
     for column in schema
     for name  = (name column)
     for value = (or (getf names-and-values name) (default-value column))
     collect name
     collect (normalize-for-column value column)))
```

It's worth defining a separate helper function, `normalize-for-column`, that takes a value and a `column` object and returns the normalized value because you'll need to perform the same normalization on query arguments.

```
(defun normalize-for-column (value column)
  (funcall (value-normalizer column) value column))
```

Now you're ready to combine this database code with code from previous chapters to build a database of data extracted from MP3 files. You can define a function, `file->row`, that uses `read-id3` from the ID3v2 library to extract an ID3 tag from a file and turns it into a plist that you can pass to `insert-row`.

```
(defun file->row (file)
  (let ((id3 (read-id3 file)))
    (list
     :file   (namestring (truename file))
```

```
    :genre  (translated-genre id3)
    :artist (artist id3)
    :album  (album id3)
    :song   (song id3)
    :track  (parse-track (track id3))
    :year   (parse-year (year id3))
    :id3-size (size id3))))
```

You don't have to worry about normalizing the values since `insert-row` takes care of that for you. You do, however, have to convert the string values returned by the `track` and `year` into numbers. The track number in an ID3 tag is sometimes stored as the ASCII representation of the track number and sometimes as a number followed by a slash followed by the total number of tracks on the album. Since you care only about the actual track number, you should use the `:end` argument to **`PARSE-INTEGER`** to specify that it should parse only up to the slash, if any.[3]

```
(defun parse-track (track)
  (when track (parse-integer track :end (position #\/ track))))

(defun parse-year (year)
  (when year (parse-integer year)))
```

Finally, you can put all these functions together, along with `walk-directory` from the portable pathnames library and `mp3-p` from the ID3v2 library, to define a function that loads an MP3 database with data extracted from all the MP3 files it can find under a given directory.

```
(defun load-database (dir db)
  (let ((count 0))
    (walk-directory
     dir
     #'(lambda (file)
         (princ #\.)
         (incf count)
         (insert-row (file->row file) db))
     :test #'mp3-p)
    (format t "~&Loaded ~d files into database." count)))
```

Querying the Database

Once you've loaded your database with data, you'll need a way to query it. For the MP3 application you'll need a slightly more sophisticated query function than you wrote in Chapter 3. This time around you want not only to be able to select rows matching particular criteria but also to limit the results to particular columns, to limit the results to unique rows, and perhaps to sort the rows by particular columns. In keeping with the spirit of relational database theory, the result of a query will be a new `table` object containing the desired rows and columns.

The query function you'll write, `select`, is loosely modeled on the `SELECT` statement from Structured Query Language (SQL). It'll take five keyword parameters: `:from`, `:columns`, `:where`, `:distinct`, and `:order-by`. The `:from` argument is the `table` object you want to query. The `:columns` argument specifies which columns should be included in the result. The value should be a list of column names, a single column name, or a `t`, the default, meaning return all columns. The `:where` argument, if provided, should be a function that accepts a row and returns true if it should be included in the results. In a moment, you'll write two functions, `matching` and `in`, that return functions appropriate for use as `:where` arguments. The `:order-by` argument, if supplied, should be a list of column names; the results will be sorted by the named columns. As with the `:columns` argument, you can specify a single column using just the name, which is equivalent to a one-item list containing the same name. Finally, the `:distinct`

argument is a boolean that says whether to eliminate duplicate rows from the results. The default value for `:distinct` is **`NIL`**.

Here are some examples of using `select`:

```
;; Select all rows where the :artist column is "Green Day"
(select :from *mp3s* :where (matching *mp3s* :artist "Green Day"))

;; Select a sorted list of artists with songs in the genre "Rock"
(select
  :columns :artist
  :from *mp3s*
  :where (matching *mp3s* :genre "Rock")
  :distinct t
  :order-by :artist)
```

The implementation of `select` with its immediate helper functions looks like this:

```
(defun select (&key (columns t) from where distinct order-by)
  (let ((rows (rows from))
        (schema (schema from)))

    (when where
      (setf rows (restrict-rows rows where)))

    (unless (eql columns 't)
      (setf schema (extract-schema (mklist columns) schema))
      (setf rows (project-columns rows schema)))

    (when distinct
      (setf rows (distinct-rows rows schema)))

    (when order-by
      (setf rows (sorted-rows rows schema (mklist order-by))))

    (make-instance 'table :rows rows :schema schema)))

(defun mklist (thing)
  (if (listp thing) thing (list thing)))

(defun extract-schema (column-names schema)
  (loop for c in column-names collect (find-column c schema)))

(defun find-column (column-name schema)
  (or (find column-name schema :key #'name)
      (error "No column: ~a in schema: ~a" column-name schema)))

(defun restrict-rows (rows where)
  (remove-if-not where rows))

(defun project-columns (rows schema)
  (map 'vector (extractor schema) rows))
```

```
(defun distinct-rows (rows schema)
  (remove-duplicates rows :test (row-equality-tester schema)))

(defun sorted-rows (rows schema order-by)
  (sort (copy-seq rows) (row-comparator order-by schema)))
```

Of course, the really interesting part of `select` is how you implement the functions `extractor`, `row-equality-tester`, and `row-comparator`.

As you can tell by how they're used, each of these functions must return a function. For instance, `project-columns` uses the value returned by `extractor` as the function argument to **`MAP`**. Since the purpose of `project-columns` is to return a set of rows with only certain column values, you can infer that `extractor` returns a function that takes a row as an argument and returns a new row containing only the columns specified in the schema it's passed. Here's how you can implement it:

```
(defun extractor (schema)
  (let ((names (mapcar #'name schema)))
    #'(lambda (row)
        (loop for c in names collect c collect (getf row c)))))
```

Note how you can do the work of extracting the names from the schema outside the body of the closure: since the closure will be called many times, you want it to do as little work as possible each time it's called.

The functions `row-equality-tester` and `row-comparator` are implemented in a similar way. To decide whether two rows are equivalent, you need to apply the appropriate equality predicate for each column to the appropriate column values. Recall from Chapter 22 that the **`LOOP`** clause `always` will return **`NIL`** as soon as a pair of values fails their test or will cause the **`LOOP`** to return **`T`**.

```
(defun row-equality-tester (schema)
  (let ((names (mapcar #'name schema))
        (tests (mapcar #'equality-predicate schema)))
    #'(lambda (a b)
        (loop for name in names and test in tests
           always (funcall test (getf a name) (getf b name))))))
```

Ordering two rows is a bit more complex. In Lisp, comparator functions return true if their first argument should be sorted ahead of the second and `NIL` otherwise. Thus, a `NIL` can mean that the second argument should be sorted ahead of the first *or* that they're equivalent. You want your row comparators to behave the same way: return `T` if the first row should be sorted ahead of the second and `NIL` otherwise.

Thus, to compare two rows, you should compare the values from the columns you're sorting by, in order, using the appropriate comparator for each column. First call the comparator with the value from the first row as the first argument. If the comparator returns true, that means the first row should definitely be sorted ahead of the second row, so you can immediately return `T`.

But if the column comparator returns `NIL`, then you need to determine whether that's because the second value should sort ahead of the first value or because they're equivalent. So you should call the comparator again with the arguments reversed. If the comparator returns true this time, it means the second column value sorts ahead of the first and thus the second row ahead of the first row, so you can return `NIL` immediately. Otherwise, the column values are equivalent, and you need to move onto the next column. If you get through all the columns without one row's value ever winning the comparison, then the rows are equivalent, and you return `NIL`. A function that implements this algorithm looks like this:

```
(defun row-comparator (column-names schema)
  (let ((comparators (mapcar #'comparator (extract-schema column-names schema))))
    #'(lambda (a b)
        (loop
           for name in column-names
           for comparator in comparators
           for a-value = (getf a name)
           for b-value = (getf b name)
           when (funcall comparator a-value b-value) return t
           when (funcall comparator b-value a-value) return nil
           finally (return nil)))))
```

Matching Functions

The `:where` argument to `select` can be any function that takes a row object and returns true if it should be included in the results. In practice, however, you'll rarely need the full power of arbitrary code to express query criteria. So you should provide two functions, `matching` and `in`, that will build query functions that allow you to express the common kinds of queries and that take care of using the proper equality predicates and value normalizers for each column.

The workhouse query-function constructor will be `matching`, which returns a function that will match rows with specific column values. You saw how it was used in the earlier examples of `select`. For instance, this call to `matching`:

```
(matching *mp3s* :artist "Green Day")
```

returns a function that matches rows whose `:artist` value is "Green Day". You can also pass multiple names and values; the returned function matches when all the columns match. For example, the following returns a closure that matches rows where the artist is "Green Day" and the album is "American Idiot":

```
(matching *mp3s* :artist "Green Day" :album "American Idiot")
```

You have to pass `matching` the table object because it needs access to the table's schema in order to get at the equality predicates and value normalizer functions for the columns it matches against.

You build up the function returned by `matching` out of smaller functions, each responsible for matching one column's value. To

build these functions, you should define a function, `column-matcher`, that takes a `column` object and an unnormalized value you want to match and returns a function that accepts a single row and returns true when the value of the given column in the row matches the normalized version of the given value.

```
(defun column-matcher (column value)
  (let ((name (name column))
        (predicate (equality-predicate column))
        (normalized (normalize-for-column value column)))
    #'(lambda (row) (funcall predicate (getf row name) normalized))))
```

You then build a list of column-matching functions for the names and values you care about with the following function, `column-matchers`:

```
(defun column-matchers (schema names-and-values)
  (loop for (name value) on names-and-values by #'cddr
     when value collect
       (column-matcher (find-column name schema) value)))
```

Now you can implement `matching`. Again, note that you do as much work as possible outside the closure in order to do it only once rather than once per row in the table.

```
(defun matching (table &rest names-and-values)
  "Build a where function that matches rows with the given column values."
  (let ((matchers (column-matchers (schema table) names-and-values)))
    #'(lambda (row)
        (every #'(lambda (matcher) (funcall matcher row)) matchers))))
```

This function is a bit of a twisty maze of closures, but it's worth contemplating for a moment to get a flavor of the possibilities of programming with functions as first-class objects.

The job of `matching` is to return a function that will be invoked on each row in a table to determine whether it should be included in the new table. So, `matching` returns a closure with one parameter, `row`.

Now recall that the function **EVERY** takes a predicate function as its first argument and returns true if, and only if, that function returns true each time it's applied to an element of the list passed as **EVERY**'s second argument. However, in this case, the list you pass to **EVERY** is itself a list of functions, the column matchers. What you want to know is that every column matcher, when invoked on the row you're currently testing, returns true. So, as the predicate argument to **EVERY**, you pass yet another closure that **FUNCALL**s the column matcher, passing it the row.

Another matching function that you'll occasionally find useful is `in`, which returns a function that matches rows where a particular column is in a given set of values. You'll define `in` to take two arguments: a column name and a table that contains the values you want to match. For instance, suppose you wanted to find all the songs in the MP3 database that have names the same as a song performed by the Dixie Chicks. You can write that where clause using `in` and a sub`select` like this:[4]

```
(select
  :columns '(:artist :song)
  :from *mp3s*
  :where (in :song
             (select
               :columns :song
               :from *mp3s*
               :where (matching *mp3s* :artist "Dixie Chicks"))))
```

Although the queries are more complex, the definition of `in` is much simpler than that of `matching`.

```
(defun in (column-name table)
  (let ((test (equality-predicate (find-column column-name (schema table))))
        (values (map 'list #'(lambda (r) (getf r column-name)) (rows table))))
    #'(lambda (row)
        (member (getf row column-name) values :test test))))
```

Getting at the Results

Since `select` returns another `table`, you need to think a bit about how you want to get at the individual row and column values in a table. If you're sure you'll never want to change the way you represent the data in a table, you can just make the structure of a table part of the API--that `table` has a slot `rows` that's a vector of plists--and use all the normal Common Lisp functions for manipulating vectors and plists to get at the values in the table. But that representation is really an internal detail that you might want to change. Also, you don't necessarily want other code manipulating the data structures directly--for instance, you don't want anyone to use **SETF** to put an unnormalized column value into a row. So it might be a good idea to define a few abstractions that provide the operations you want to support. Then if you decide to change the internal representation later, you'll need to change only the implementation of these functions and macros. And while Common Lisp doesn't enable you to absolutely prevent folks from getting at "internal" data, by providing an official API you at least make it clear where the boundary is.

Probably the most common thing you'll need to do with the results of a query is to iterate over the individual rows and extract specific column values. So you need to provide a way to do both those things without touching the `rows` vector directly or using **GETF** to get at the column values within a row.

For now these operations are trivial to implement; they're merely wrappers around the code you'd write if you didn't have these

abstractions. You can provide two ways to iterate over the rows of a table: a macro `do-rows`, which provides a basic looping construct, and a function `map-rows`, which builds a list containing the results of applying a function to each row in the table.[5]

```
(defmacro do-rows ((row table) &body body)
  `(loop for ,row across (rows ,table) do ,@body))

(defun map-rows (fn table)
  (loop for row across (rows table) collect (funcall fn row)))
```

To get at individual column values within a row, you should provide a function, `column-value`, that takes a row and a column name and returns the appropriate value. Again, it's a trivial wrapper around the code you'd write otherwise. But if you change the internal representation of a table later, users of `column-value` needn't be any the wiser.

```
(defun column-value (row column-name)
  (getf row column-name))
```

While `column-value` is a sufficient abstraction for getting at column values, you'll often want to get at the values of multiple columns at once. So you can provide a bit of syntactic sugar, a macro, `with-column-values`, that binds a set of variables to the values extracted from a row using the corresponding keyword names. Thus, instead of writing this:

```
(do-rows (row table)
  (let ((song (column-value row :song))
        (artist (column-value row :artist))
        (album (column-value row :album)))
    (format t "~a by ~a from ~a~%" song artist album)))
```

you can simply write the following:

```
(do-rows (row table)
  (with-column-values (song artist album) row
    (format t "~a by ~a from ~a~%" song artist album)))
```

Again, the actual implementation isn't complicated if you use the `once-only` macro from Chapter 8.

```
(defmacro with-column-values ((&rest vars) row &body body)
  (once-only (row)
    `(let ,(column-bindings vars row) ,@body)))

(defun column-bindings (vars row)
  (loop for v in vars collect `(,v (column-value ,row ,(as-keyword v)))))

(defun as-keyword (symbol)
  (intern (symbol-name symbol) :keyword))
```

Finally, you should provide abstractions for getting at the number of rows in a table and for accessing a specific row by numeric index.

```
(defun table-size (table)
  (length (rows table)))

(defun nth-row (n table)
  (aref (rows table) n))
```

Other Database Operations

Finally, you'll implement a few other database operations that you'll need in Chapter 29. The first two are analogs of the SQL `DELETE` statement. The function `delete-rows` is used to delete rows from a table that match particular criteria. Like `select`, it takes `:from` and `:where` keyword arguments. Unlike `select`, it doesn't return a new table--it actually modifies the table passed as the `:from` argument.

```
(defun delete-rows (&key from where)
  (loop
     with rows = (rows from)
     with store-idx = 0
     for read-idx from 0
     for row across rows
     do (setf (aref rows read-idx) nil)
     unless (funcall where row) do
       (setf (aref rows store-idx) row)
       (incf store-idx)
     finally (setf (fill-pointer rows) store-idx)))
```

In the interest of efficiency, you might want to provide a separate function for deleting all the rows from a table.

```
(defun delete-all-rows (table)
  (setf (rows table) (make-rows *default-table-size*)))
```

The remaining table operations don't really map to normal relational database operations but will be useful in the MP3 browser application. The first is a function to sort the rows of a table in place.

```
(defun sort-rows (table &rest column-names)
  (setf (rows table) (sort (rows table) (row-comparator column-names (schema table))))
  table)
```

On the flip side, in the MP3 browser application, you'll need a function that shuffles a table's rows in place using the function `nshuffle-vector` from Chapter 23.

```
(defun shuffle-table (table)
  (nshuffle-vector (rows table))
  table)
```

And finally, again for the purposes of the MP3 browser, you should provide a function that selects *n* random rows, returning the results as a new table. It also

uses `nshuffle-vector` along with a version of `random-sample` based on Algorithm S from Donald Knuth's *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Third Edition (Addison-Wesley, 1998) that I discussed in Chapter 20.

```
(defun random-selection (table n)
  (make-instance
   'table
   :schema (schema table)
   :rows (nshuffle-vector (random-sample (rows table) n))))

(defun random-sample (vector n)
  "Based on Algorithm S from Knuth. TAOCP, vol. 2. p. 142"
  (loop with selected = (make-array n :fill-pointer 0)
     for idx from 0
     do
       (loop
          with to-select = (- n (length selected))
          for remaining = (- (length vector) idx)
          while (>= (* remaining (random 1.0)) to-select)
          do (incf idx))
       (vector-push (aref vector idx) selected)
     when (= (length selected) n) return selected))
```

With this code you'll be ready, in Chapter 29, to build a Web interface for browsing a collection of MP3 files. But before you get to that, you need to implement the part of the server that streams MP3s using the Shoutcast protocol, which is the topic of the next chapter.

---

[1]The general theory behind interning objects is that if you're going to compare a particular value many times, it's worth it to pay the cost of interning it. The `value-normalizer` runs once when you insert a value into the table and, as you'll see, once at the beginning of each query. Since a query can involve invoking the `equality-predicate` once per row in the table, the amortized cost of interning the values will quickly approach zero.

[2]As always, the first causality of concise exposition in programming books is proper error handling; in production code you'd probably want to define your own error type, such as the following, and signal it instead:

```
(error 'illegal-column-value :value value :column column)
```

Then you'd want to think about where you can add restarts that might be able to recover from this condition. And, finally, in any given application you could establish condition handlers that would choose from among those restarts.

[3]If any MP3 files have malformed data in the track and year frames, **`PARSE-INTEGER`** could signal an error. One way to deal with that is to pass **`PARSE-INTEGER`** the `:junk-allowed` argument of **`T`**, which will cause it to ignore any non-numeric junk following the number and to return **`NIL`** if no number can be found in the string. Or, if you want practice at using the condition system, you could define an error and signal it from

these functions when the data is malformed and also establish a few restarts to allow these functions to recover.

[4]This query will also return all the songs performed by the Dixie Chicks. If you want to limit it to songs by artists other than the Dixie Chicks, you need a more complex `:where` function. Since the `:where` argument can be any function, it's certainly possible; you could remove the Dixie Chicks' own songs with this query:

```
(let* ((dixie-chicks (matching *mp3s* :artist "Dixie Chicks"))
       (same-song (in :song (select :columns :song :from *mp3s* :where dixie-chicks)))
       (query #'(lambda (row) (and (not (funcall dixie-chicks row)) (funcall same-song row)))))
  (select :columns '(:artist :song) :from *mp3s* :where query))
```

This obviously isn't quite as convenient. If you were going to write an application that needed to do lots of complex queries, you might want to consider coming up with a more expressive query language.

[5]The version of LOOP implemented at M.I.T. before Common Lisp was standardized included a mechanism for extending the LOOP grammar to support iteration over new data structures. Some Common Lisp implementations that inherited their LOOP implementation from that code base may still support that facility, which would make `do-rows` and `map-rows` less necessary.

Copyright © 2003-2005, Peter Seibel

28. Practical: A Shoutcast Server

In this chapter you'll develop another important part of what will eventually be a Web-based application for streaming MP3s, namely, the server that implements the Shoutcast protocol for actually streaming MP3s to clients such as iTunes, XMMS,[1] or Winamp.

The Shoutcast Protocol

The Shoutcast protocol was invented by the folks at Nullsoft, the makers of the Winamp MP3 software. It was designed to support Internet audio broadcasting--Shoutcast DJs send audio data from their personal computers to a central Shoutcast server that then turns around and streams it out to any connected listeners.

The server you'll build is actually only half a true Shoutcast server--you'll use the protocol that Shoutcast servers use to stream MP3s to listeners, but your server will be able to serve only songs already stored on the file system of the computer where the server is running.

You need to worry about only two parts of the Shoutcast protocol: the request that a client makes in order to start receiving a stream and the format of the response, including the mechanism by which metadata about what song is currently playing is embedded in the stream.

The initial request from the MP3 client to the Shoutcast server is formatted as a normal HTTP request. In response, the Shoutcast server sends an ICY response that looks like an HTTP response except with the string "ICY"[2] in place of the normal HTTP version string and with different headers. After sending the headers and a blank line, the server streams a potentially endless amount of MP3 data.

The only tricky thing about the Shoutcast protocol is the way metadata about the songs being streamed is embedded in the data sent to the client. The problem facing the Shoutcast designers was to provide a way for the Shoutcast server to communicate new title information to the client each time it started playing a new song so the client could display it in its UI. (Recall from Chapter 25 that the MP3 format doesn't make any provision for encoding metadata.) While one of the design goals of ID3v2 had been to make it better suited for use when streaming MP3s, the Nullsoft folks decided to go their own route and invent a new scheme that's fairly easy to implement on both the client side and the server side. That, of course, was ideal for them since they were also the authors of their own MP3 client.

Their scheme was to simply ignore the structure of MP3 data and embed a chunk of self-delimiting metadata every *n* bytes. The client would then be responsible for stripping out this metadata so it wasn't treated as MP3 data. Since metadata sent to a client that isn't ready for it will cause glitches in the sound, the server is supposed to send metadata only if the client's original request contains a special Icy-Metadata header. And in order for the client to know how often to expect metadata, the server must send back a header Icy-Metaint whose value is the number of bytes of MP3 data that will be sent between each chunk of metadata.

The basic content of the metadata is a string of the form "StreamTitle='*title*';" where *title* is the title of the current song and can't contain single quote marks. This payload is encoded as a length-delimited array of bytes: a single byte is sent indicating how

many 16-byte blocks follow, and then that many blocks are sent. They contain the string payload as an ASCII string, with the final block padded out with null bytes as necessary.

Thus, the smallest legal metadata chunk is a single byte, zero, indicating zero subsequent blocks. If the server doesn't need to update the metadata, it can send such an empty chunk, but it must send at least the one byte so the client doesn't throw away actual MP3 data.

Song Sources

Because a Shoutcast server has to keep streaming songs to the client for as long as it's connected, you need to provide your server with a source of songs to draw on. In the Web-based application, each connected client will have a playlist that can be manipulated via the Web interface. But in the interest of avoiding excessive coupling, you should define an interface that the Shoutcast server can use to obtain songs to play. You can write a simple implementation of this interface now and then a more complex one as part of the Web application you'll build in Chapter 29.

The Package

The package for the code you'll develop in this chapter looks like this:

```
(defpackage :com.gigamonkeys.shoutcast
  (:use :common-lisp
        :net.aserve
        :com.gigamonkeys.id3v2)
  (:export :song
           :file
           :title
           :id3-size
           :find-song-source
           :current-song
           :still-current-p
           :maybe-move-to-next-song
           :*song-source-type*))
```

The idea behind the interface is that the Shoutcast server will find a source of songs based on an ID extracted from the AllegroServe request object. It can then do three things with the song source it's given.

- Get the current song from the source
- Tell the song source that it's done with the current song
- Ask the source whether the song it was given earlier is still the current song

The last operation is necessary because there may be ways--and will be in Chapter 29--to manipulate the songs source outside the Shoutcast server. You can express the operations the Shoutcast server needs with the following generic functions:

```
(defgeneric current-song (source)
  (:documentation "Return the currently playing song or NIL."))

(defgeneric maybe-move-to-next-song (song source)
  (:documentation
   "If the given song is still the current one update the value
returned by current-song."))

(defgeneric still-current-p (song source)
  (:documentation
   "Return true if the song given is the same as the current-song."))
```

The function `maybe-move-to-next-song` is defined the way it is so a single operation checks whether the song is current and, if it is, moves the song source to the next song. This will be important in the next chapter when you need to implement a song source that can be safely manipulated from two different threads.[3]

To represent the information about a song that the Shoutcast server needs, you can define a class, `song`, with slots to hold the name of the MP3 file, the title to send in the Shoutcast metadata, and the size of the ID3 tag so you can skip it when serving up the file.

```
(defclass song ()
  ((file     :reader file     :initarg :file)
   (title    :reader title    :initarg :title)
   (id3-size :reader id3-size :initarg :id3-size)))
```

The value returned by `current-song` (and thus the first argument to `still-current-p` and `maybe-move-to-next-song`) will be an instance of `song`.

In addition, you need to define a generic function that the server can use to find a song source based on the type of source desired and the request object. Methods will specialize the `type` parameter in order to return different kinds of song source and will pull whatever information they need from the request object to determine which source to return.

```
(defgeneric find-song-source (type request)
  (:documentation "Find the song-source of the given type for the given request."))
```

However, for the purposes of this chapter, you can use a trivial implementation of this interface that always uses the same object, a simple queue of song objects that you can manipulate from the REPL. You can start by defining a class, `simple-song-queue`, and a global variable, `*songs*`, that holds an instance of this class.

```
(defclass simple-song-queue ()
  ((songs :accessor songs :initform (make-array 10 :adjustable t :fill-pointer 0))
   (index :accessor index :initform 0)))

(defparameter *songs* (make-instance 'simple-song-queue))
```

Then you can define a method on `find-song-source` that specializes `type` with an **`EQL`** specializer on the symbol `singleton` and returns the instance stored in `*songs*`.

```
(defmethod find-song-source ((type (eql 'singleton)) request)
  (declare (ignore request))
  *songs*)
```

Now you just need to implement methods on the three generic functions that the Shoutcast server will use.

```
(defmethod current-song ((source simple-song-queue))
  (when (array-in-bounds-p (songs source) (index source))
    (aref (songs source) (index source))))

(defmethod still-current-p (song (source simple-song-queue))
  (eql song (current-song source)))

(defmethod maybe-move-to-next-song (song (source simple-song-queue))
  (when (still-current-p song source)
    (incf (index source))))
```

And for testing purposes you should provide a way to add songs to this queue.

```
(defun add-file-to-songs (file)
  (vector-push-extend (file->song file) (songs *songs*)))

(defun file->song (file)
  (let ((id3 (read-id3 file)))
    (make-instance
     'song
     :file (namestring (truename file))
     :title (format nil "~a by ~a from ~a" (song id3) (artist id3) (album id3))
     :id3-size (size id3))))
```

Implementing Shoutcast

Now you're ready to implement the Shoutcast server. Since the Shoutcast protocol is loosely based on HTTP, you can implement the server as a function within AllegroServe. However, since you need to interact with some of the low-level features of AllegroServe, you can't use the `define-url-function` macro from Chapter 26. Instead, you need to write a regular function that looks like this:

```
(defun shoutcast (request entity)
  (with-http-response
      (request entity :content-type "audio/MP3" :timeout *timeout-seconds*)
    (prepare-icy-response request *metadata-interval*)
    (let ((wants-metadata-p (header-slot-value request :icy-metadata)))
      (with-http-body (request entity)
        (play-songs
         (request-socket request)
         (find-song-source *song-source-type* request)
         (if wants-metadata-p *metadata-interval*))))))
```

Then publish that function under the path `/stream.mp3` like this:[4]

```
(publish :path "/stream.mp3" :function 'shoutcast)
```

In the call to `with-http-response`, in addition to the usual `request` and `entity` arguments, you need to pass `:content-type` and `:timeout` arguments. The `:content-type` argument tells AllegroServe how to set the Content-Type header it sends. And the `:timeout` argument specifies the number of seconds AllegroServe gives the function to generate its response. By default AllegroServe times out each request after five minutes. Because you're going to stream an essentially endless sequence of MP3s, you need much more time. There's no way to tell AllegroServe to *never* time out the request, so you should set it to the value of `*timeout-seconds*`, which you can

define to some suitably large value such as the number of seconds in ten years.

```
(defparameter *timeout-seconds* (* 60 60 24 7 52 10))
```

Then, within the body of the `with-http-response` and before the call to `with-http-body` that will cause the response headers to be sent, you need to manipulate the reply that AllegroServe will send. The function `prepare-icy-response` encapsulates the necessary manipulations: changing the protocol string from the default of "HTTP" to "ICY" and adding the Shoutcast-specific headers.[5] You also need, in order to work around a bug in iTunes, to tell AllegroServe not to use *chunked transfer-encoding*.[6] The functions `request-reply-protocol-string`, `request-uri`, and `reply-header-slot-value` are all part of AllegroServe.

```
(defun prepare-icy-response (request metadata-interval)
  (setf (request-reply-protocol-string request) "ICY")
  (loop for (k v) in (reverse
       `((:|icy-metaint| ,(princ-to-string metadata-interval))
         (:|icy-notice1| "<BR>This stream blah blah blah<BR>")
         (:|icy-notice2| "More blah")
         (:|icy-name|    "MyLispShoutcastServer")
         (:|icy-genre|   "Unknown")
         (:|icy-url|     ,(request-uri request))
         (:|icy-pub|     "1")))
     do (setf (reply-header-slot-value request k) v))
  ;; iTunes, despite claiming to speak HTTP/1.1, doesn't understand
  ;; chunked Transfer-encoding. Grrr. So we just turn it off.
  (turn-off-chunked-transfer-encoding request))

(defun turn-off-chunked-transfer-encoding (request)
  (setf (request-reply-strategy request)
        (remove :chunked (request-reply-strategy request))))
```

Within the `with-http-body` of `shoutcast`, you actually stream the MP3 data. The function `play-songs` takes the stream to which it should write the data, the song source, and the metadata interval it should use or **`NIL`** if the client doesn't want metadata. The stream is the socket obtained from the request object, the song source is obtained

by calling `find-song-source`, and the metadata interval comes from the global variable `*metadata-interval*`. The type of song source is controlled by the variable `*song-source-type*`, which for now you can set to `singleton` in order to use the `simple-song-queue` you implemented previously.

```
(defparameter *metadata-interval* (expt 2 12))

(defparameter *song-source-type* 'singleton)
```

The function `play-songs` itself doesn't do much--it loops calling the function `play-current`, which does all the heavy lifting of sending the contents of a single MP3 file, skipping the ID3 tag and embedding ICY metadata. The only wrinkle is that you need to keep track of when to send the metadata.

Since you must send metadata chunks at a fixed intervals, regardless of when you happen to switch from one MP3 file to the next, each time you call `play-current` you need to tell it when the next metadata is due, and when it returns, it must tell you the same thing so you can pass the information to the next call to `play-current`. If `play-current` gets **`NIL`** from the song source, it returns **`NIL`**, which allows the `play-songs` **`LOOP`** to end.

In addition to handling the looping, `play-songs` also provides a **`HANDLER-CASE`** to trap the error that will be signaled when the MP3 client disconnects from the server and one of the writes to the socket, down in `play-current`, fails. Since the **`HANDLER-CASE`** is outside the **`LOOP`**, handling the error will break out of the loop, allowing `play-songs` to return.

```
(defun play-songs (stream song-source metadata-interval)
  (handler-case
```

```
      (loop
         for next-metadata = metadata-interval
         then (play-current
               stream
               song-source
               next-metadata
               metadata-interval)
         while next-metadata)
    (error (e) (format *trace-output* "Caught error in play-songs: ~a" e))))
```

Finally, you're ready to implement `play-current`, which actually sends the Shoutcast data. The basic idea is that you get the current song from the song source, open the song's file, and then loop reading data from the file and writing it to the socket until either you reach the end of the file or the current song is no longer the current song.

There are only two complications: One is that you need to make sure you send the metadata at the correct interval. The other is that if the file starts with an ID3 tag, you want to skip it. If you don't worry too much about I/O efficiency, you can implement `play-current` like this:

```
(defun play-current (out song-source next-metadata metadata-interval)
  (let ((song (current-song song-source)))
    (when song
      (let ((metadata (make-icy-metadata (title song))))
        (with-open-file (mp3 (file song))
          (unless (file-position mp3 (id3-size song))
            (error "Can't skip to position ~d in ~a" (id3-size song) (file song)))
          (loop for byte = (read-byte mp3 nil nil)
             while (and byte (still-current-p song song-source)) do
               (write-byte byte out)
               (decf next-metadata)
             when (and (zerop next-metadata) metadata-interval) do
               (write-sequence metadata out)
               (setf next-metadata metadata-interval))

          (maybe-move-to-next-song song song-source)))
      next-metadata)))
```

This function gets the current song from the song source and gets a buffer containing the metadata it'll need to send by passing the title to `make-icy-metadata`. Then it opens the file and skips past the ID3 tag using the two-argument form of **`FILE-POSITION`**. Then it

commences reading bytes from the file and writing them to the request stream.[7]

It'll break out of the loop either when it reaches the end of the file or when the song source's current song changes out from under it. In the meantime, whenever `next-metadata` gets to zero (if you're supposed to send metadata at all), it writes `metadata` to the stream and resets `next-metadata`. Once it finishes the loop, it checks to see if the song is still the song source's current song; if it is, that means it broke out of the loop because it read the whole file, in which case it tells the song source to move to the next song. Otherwise, it broke out of the loop because someone changed the current song out from under it, and it just returns. In either case, it returns the number of bytes left before the next metadata is due so it can be passed in the next call to `play-current`.[8]

The function `make-icy-metadata`, which takes the title of the current song and generates an array of bytes containing a properly formatted chunk of ICY metadata, is also straightforward.[9]

```
(defun make-icy-metadata (title)
  (let* ((text (format nil "StreamTitle='~a';" (substitute #\Space #\' title)))
         (blocks (ceiling (length text) 16))
         (buffer (make-array (1+ (* blocks 16))
                             :element-type '(unsigned-byte 8)
                             :initial-element 0)))
    (setf (aref buffer 0) blocks)
    (loop
       for char across text
       for i from 1
       do (setf (aref buffer i) (char-code char)))
    buffer))
```

Depending on how your particular Lisp implementation handles its streams, and also how many MP3 clients you want to serve at once,

the simple version of `play-current` may or may not be efficient enough.

The potential problem with the simple implementation is that you have to call **`READ-BYTE`** and **`WRITE-BYTE`** for every byte you transfer. It's possible that each call may result in a relatively expensive system call to read or write one byte. And even if Lisp implements its own streams with internal buffering so not every call to **`READ-BYTE`** or **`WRITE-BYTE`** results in a system call, function calls still aren't free. In particular, in implementations that provide user-extensible streams using so-called Gray Streams, **`READ-BYTE`** and **`WRITE-BYTE`** may result in a generic function call under the covers to dispatch on the class of the stream argument. While generic function dispatch is normally speedy enough that you don't have to worry about it, it's a bit more expensive than a nongeneric function call and thus not something you necessarily want to do several million times in a few minutes if you can avoid it.

A more efficient, if slightly more complex, way to implement `play-current` is to read and write multiple bytes at a time using the functions **`READ-SEQUENCE`** and **`WRITE-SEQUENCE`**. This also gives you a chance to match your file reads with the natural block size of the file system, which will likely give you the best disk throughput. Of course, no matter what buffer size you use, keeping track of when to send the metadata becomes a bit more complicated. A more efficient version of `play-current` that uses **`READ-SEQUENCE`** and **`WRITE-SEQUENCE`** might look like this:

```
(defun play-current (out song-source next-metadata metadata-interval)
  (let ((song (current-song song-source)))
    (when song
```

```
(let ((metadata (make-icy-metadata (title song)))
      (buffer (make-array size :element-type '(unsigned-byte 8))))
  (with-open-file (mp3 (file song))
    (labels ((write-buffer (start end)
               (if metadata-interval
                 (write-buffer-with-metadata start end)
                 (write-sequence buffer out :start start :end end)))

             (write-buffer-with-metadata (start end)
               (cond
                 ((> next-metadata (- end start))
                  (write-sequence buffer out :start start :end end)
                  (decf next-metadata (- end start)))
                 (t
                  (let ((middle (+ start next-metadata)))
                    (write-sequence buffer out :start start :end middle)
                    (write-sequence metadata out)
                    (setf next-metadata metadata-interval)
                    (write-buffer-with-metadata middle end))))))

      (multiple-value-bind (skip-blocks skip-bytes)
          (floor (id3-size song) (length buffer))

        (unless (file-position mp3 (* skip-blocks (length buffer)))
          (error "Couldn't skip over ~d ~d byte blocks."
                 skip-blocks (length buffer)))

        (loop for end = (read-sequence buffer mp3)
           for start = skip-bytes then 0
           do (write-buffer start end)
           while (and (= end (length buffer))
                      (still-current-p song song-source)))

        (maybe-move-to-next-song song song-source)))))
next-metadata)))
```

Now you're ready to put all the pieces together. In the next chapter you'll write a Web interface to the Shoutcast server developed in this chapter, using the MP3 database from Chapter 27 as the source of songs.

---

[1]The version of XMMS shipped with Red Hat 8.0 and 9.0 and Fedora no longer knows how to play MP3s because the folks at Red Hat were worried about the licensing issues related to the MP3 codec. To get an XMMS with MP3 support on these versions of Linux, you can grab the source from `http://www.xmms.org` and build it yourself. Or, see `http://www.fedorafaq.org/#xmms-mp3` for information about other possibilities.

[2]To further confuse matters, there's a different streaming protocol called *Icecast*. There seems to be no connection between the ICY header used by Shoutcast and the Icecast

protocol.

[3]Technically, the implementation in this chapter will also be manipulated from two threads--the AllegroServe thread running the Shoutcast server and the REPL thread. But you can live with the race condition for now. I'll discuss how to use locking to make code thread safe in the next chapter.

[4]Another thing you may want to do while working on this code is to evaluate the form `(net.aserve::debug-on :notrap)`. This tells AllegroServe to not trap errors signaled by your code, which will allow you to debug them in the normal Lisp debugger. In SLIME this will pop up a SLIME debugger buffer just like any other error.

[5]Shoutcast headers are usually sent in lowercase, so you need to escape the names of the keyword symbols used to identify them to AllegroServe to keep the Lisp reader from converting them to all uppercase. Thus, you'd write `:|icy-metaint|` rather than `:icy-metaint`. You could also write `:\i\c\y-\m\e\t\a\i\n\t`, but that'd be silly.

[6]The function `turn-off-chunked-transfer-encoding` is a bit of a kludge. There's no way to turn off chunked transfer encoding via AllegroServe's official APIs without specifying a content length because any client that advertises itself as an HTTP/1.1 client, which iTunes does, is supposed to understand it. But this does the trick.

[7]Most MP3-playing software will display the metadata somewhere in the user interface. However, the XMMS program on Linux by default doesn't. To get XMMS to display Shoutcast metadata, press Ctrl+P to see the Preferences pane. Then in the Audio I/O Plugins tab (the leftmost tab in version 1.2.10), select the MPEG Layer 1/2/3 Player (`libmpg123.so`) and hit the Configure button. Then select the Streaming tab on the configuration window, and at the bottom of the tab in the SHOUTCAST/Icecast section, check the "Enable SHOUTCAST/Icecast title streaming" box.

[8]Folks coming to Common Lisp from Scheme might wonder why `play-current` can't just call itself recursively. In Scheme that would work fine since Scheme implementations are required by the Scheme specification to support "an unbounded number of active tail calls." Common Lisp implementations are allowed to have this property, but it isn't required by the language standard. Thus, in Common Lisp the idiomatic way to write loops is with a looping construct, not with recursion.

[9]This function assumes, as has other code you've written, that your Lisp implementation's internal character encoding is ASCII or a superset of ASCII, so you can use **CHAR-CODE** to translate Lisp **CHARACTER** objects to bytes of ASCII data.

Copyright © 2003-2005, Peter Seibel

29. Practical: An MP3 Browser

The final step in building the MP3 streaming application is to provide a Web interface that allows a user to find the songs they want to listen to and add them to a playlist that the Shoutcast server will draw upon when the user's MP3 client requests the stream URL. For this component of the application, you'll pull together several bits of code from the previous few chapters: the MP3 database, the `define-url-function` macro from Chapter 26, and, of course, the Shoutcast server itself.

Playlists

The basic idea behind the interface will be that each MP3 client that connects to the Shoutcast server gets its own *playlist*, which serves as the source of songs for the Shoutcast server. A playlist will also provide facilities beyond those needed by the Shoutcast server: through the Web interface the user will be able to add songs to the playlist, delete songs already in the playlist, and reorder the playlist by sorting and shuffling.

You can define a class to represent playlists like this:

```
(defclass playlist ()
  ((id           :accessor id           :initarg :id)
   (songs-table  :accessor songs-table  :initform (make-playlist-table))
   (current-song :accessor current-song :initform *empty-playlist-song*)
   (current-idx  :accessor current-idx  :initform 0)
   (ordering     :accessor ordering     :initform :album)
   (shuffle      :accessor shuffle      :initform :none)
   (repeat       :accessor repeat       :initform :none)
   (user-agent   :accessor user-agent   :initform "Unknown")
   (lock         :reader   lock         :initform (make-process-lock))))
```

The `id` of a playlist is the key you extract from the request object passed to `find-song-source` when looking up a playlist. You don't actually need to store it in the `playlist` object, but it makes debugging a bit easier if you can find out from an arbitrary playlist object what its `id` is.

The heart of the playlist is the `songs-table` slot, which will hold a `table` object. The schema for this table will be the same as for the main MP3 database. The function `make-playlist-table`, which you use to initialize `songs-table`, is simply this:

```
(defun make-playlist-table ()
  (make-instance 'table :schema *mp3-schema*))
```

The Package

You can define the package for the code in this chapter with the following `DEFPACKAGE`:

```
(defpackage :com.gigamonkeys.mp3-browser
  (:use :common-lisp
        :net.aserve
        :com.gigamonkeys.html
        :com.gigamonkeys.shoutcast
        :com.gigamonkeys.url-function
        :com.gigamonkeys.mp3-database
        :com.gigamonkeys.id3v2)
  (:import-from :acl-socket
                :ipaddr-to-dotted
                :remote-host)
  (:import-from :multiprocessing
                :make-process-lock
                :with-process-lock)
  (:export :start-mp3-browser))
```

Because this is a high-level application, it uses a lot of lower-level packages. It also imports three symbols from the `ACL-SOCKET` package and two more from `MULTIPROCESSING` since it just needs those five and not the other 139 symbols those two packages export.

By storing the list of songs as a table, you can use the database functions from Chapter 27 to manipulate the playlist: you can add to the playlist with `insert-row`, delete songs with `delete-rows`, and reorder the playlist with `sort-rows` and `shuffle-table`.

The `current-song` and `current-idx` slots keep track of which song is playing: `current-song` is an actual `song` object, while `current-idx` is the index into the `songs-table` of the row representing the current song. You'll see in the section "Manipulating the Playlist" how to make sure `current-song` is updated whenever `current-idx` changes.

The `ordering` and `shuffle` slots hold information about how the songs in `songs-table` are to be ordered. The `ordering` slot holds a keyword that tells how the `songs-table` should be sorted when it's not shuffled. The legal values are `:genre`, `:artist`, `:album`, and `:song`. The `shuffle` slot holds one of the keywords `:none`, `:song`, or `:album`, which specifies how `songs-table` should be shuffled, if at all.

The `repeat` slot also holds a keyword, one of `:none`, `:song`, or `:all`, which specifies the repeat mode for the playlist. If `repeat` is `:none`, after the last song in the `songs-table` has been played, the `current-song` goes back to a default MP3. When `:repeat` is `:song`, the playlist keeps returning the same `current-song` forever. And if it's `:all`, after the last song, `current-song` goes back to the first song.

The `user-agent` slot holds the value of the User-Agent header sent by the MP3 client in its request for the stream. You need to hold onto this value purely for use in the Web interface--the User-Agent header identifies the program that made the request, so you can display the value on the page that lists all the playlists to make it easier to tell which playlist goes with which connection when multiple clients connect.

Finally, the `lock` slot holds a *process lock* created with the function `make-process-lock`, which is part of Allegro's `MULTIPROCESSING` package. You'll need to use that lock in certain functions that manipulate `playlist` objects to ensure that only one thread at a time manipulates a given playlist object. You can define the following macro, built upon the `with-process-lock` macro from

`MULTIPROCESSING`, to give an easy way to wrap a body of code that should be performed while holding a playlist's lock:

```
(defmacro with-playlist-locked ((playlist) &body body)
  `(with-process-lock ((lock ,playlist))
     ,@body))
```

The `with-process-lock` macro acquires exclusive access to the process lock given and then executes the body forms, releasing the lock afterward. By default, `with-process-lock` allows recursive locks, meaning the same thread can safely acquire the same lock multiple times.

Playlists As Song Sources

To use `playlist`s as a source of songs for the Shoutcast server, you'll need to implement a method on the generic function `find-song-source` from Chapter 28. Since you're going to have multiple playlists, you need a way to find the right one for each client that connects to the server. The mapping part is easy--you can define a variable that holds an **EQUAL** hash table that you can use to map from some identifier to the `playlist` object.

```
(defvar *playlists* (make-hash-table :test #'equal))
```

You'll also need to define a process lock to protect access to this hash table like this:

```
(defparameter *playlists-lock* (make-process-lock :name "playlists-lock"))
```

Then define a function that looks up a playlist given an ID, creating a new `playlist` object if necessary and using `with-process-lock` to ensure that only one thread at a time manipulates the hash table.[1]

```
(defun lookup-playlist (id)
  (with-process-lock (*playlists-lock*)
    (or (gethash id *playlists*)
        (setf (gethash id *playlists*) (make-instance 'playlist :id id)))))
```

Then you can implement `find-song-source` on top of that function and another, `playlist-id`, that takes an AllegroServe request object and returns the appropriate playlist identifier. The `find-song-source` function is also where you grab the User-Agent string out of the request object and stash it in the playlist object.

```
(defmethod find-song-source ((type (eql 'playlist)) request)
  (let ((playlist (lookup-playlist (playlist-id request))))
    (with-playlist-locked (playlist)
      (let ((user-agent (header-slot-value request :user-agent)))
```

```
      (when user-agent (setf (user-agent playlist) user-agent))))
    playlist))
```

The trick, then, is how you implement `playlist-id`, the function that extracts the identifier from the request object. You have a couple options, each with different implications for the user interface. You can pull whatever information you want out of the request object, but however you decide to identify the client, you need some way for the user of the Web interface to get hooked up to the right playlist.

For now you can take an approach that "just works" as long as there's only one MP3 client per machine connecting to the server and as long as the user is browsing the Web interface from the machine running the MP3 client: you'll use the IP address of the client machine as the identifier. This way you can find the right playlist for a request regardless of whether the request is from the MP3 client or a Web browser. You will, however, provide a way in the Web interface to select a different playlist from the browser, so the only real constraint this choice puts on the application is that there can be only one connected MP3 client per client IP address.[2] The implementation of `playlist-id` looks like this:

```
(defun playlist-id (request)
  (ipaddr-to-dotted (remote-host (request-socket request))))
```

The function `request-socket` is part of AllegroServe, while `remote-host` and `ipaddr-to-dotted` are part of Allegro's socket library.

To make a playlist usable as a song source by the Shoutcast server, you need to define methods on `current-song`, `still-current-p`, and `maybe-move-to-next-song` that specialize their `source` parameter on `playlist`. The `current-song` method is already taken care of: by

defining the accessor `current-song` on the eponymous slot, you automatically got a `current-song` method specialized on `playlist` that returns the value of that slot. However, to make accesses to the `playlist` thread safe, you need to lock the `playlist` before accessing the `current-song` slot. In this case, the easiest way is to define an `:around` method like the following:

```
(defmethod current-song :around ((playlist playlist))
  (with-playlist-locked (playlist) (call-next-method)))
```

Implementing `still-current-p` is also quite simple, assuming you can be sure that `current-song` gets updated with a new `song` object only when the current song actually changes. Again, you need to acquire the process lock to ensure you get a consistent view of the `playlist`'s state.

```
(defmethod still-current-p (song (playlist playlist))
  (with-playlist-locked (playlist)
    (eql song (current-song playlist))))
```

The trick, then, is to make sure the `current-song` slot gets updated at the right times. However, the current song can change in a number of ways. The obvious one is when the Shoutcast server calls `maybe-move-to-next-song`. But it can also change when songs are added to the playlist, when the Shoutcast server has run out of songs, or even if the playlist's repeat mode is changed.

Rather than trying to write code specific to every situation to determine whether to update `current-song`, you can define a function, `update-current-if-necessary`, that updates `current-song` if the `song` object in `current-song` no longer matches the file that the `current-idx` slot says should be playing. Then, if you call this function after any manipulation of the playlist that could possibly put

those two slots out of sync, you're sure to keep `current-song` set properly. Here are `update-current-if-necessary` and its helper functions:

```
(defun update-current-if-necessary (playlist)
  (unless (equal (file (current-song playlist))
                 (file-for-current-idx playlist))
    (reset-current-song playlist)))

(defun file-for-current-idx (playlist)
  (if (at-end-p playlist)
    nil
    (column-value (nth-row (current-idx playlist) (songs-table playlist)) :file)))

(defun at-end-p (playlist)
  (>= (current-idx playlist) (table-size (songs-table playlist))))
```

You don't need to add locking to these functions since they'll be called only from functions that will take care of locking the playlist first.

The function `reset-current-song` introduces one more wrinkle: because you want the playlist to provide an endless stream of MP3s to the client, you don't want to ever set `current-song` to **`NIL`**. Instead, when a playlist runs out of songs to play--when `songs-table` is empty or after the last song has been played and `repeat` is set to `:none`--then you need to set `current-song` to a special song whose file is an MP3 of silence[3] and whose title explains why no music is playing. Here's some code to define two parameters, `*empty-playlist-song*` and `*end-of-playlist-song*`, each set to a song with the file named by `*silence-mp3*` as their file and an appropriate title:

```
(defparameter *silence-mp3* ...)

(defun make-silent-song (title &optional (file *silence-mp3*))
  (make-instance
   'song
   :file file
   :title title
   :id3-size (if (id3-p file) (size (read-id3 file)) 0)))
```

```
(defparameter *empty-playlist-song* (make-silent-song "Playlist empty."))

(defparameter *end-of-playlist-song* (make-silent-song "At end of playlist."))
```

`reset-current-song` uses these parameters when the `current-idx` doesn't point to a row in `songs-table`. Otherwise, it sets `current-song` to a `song` object representing the current row.

```
(defun reset-current-song (playlist)
  (setf
   (current-song playlist)
   (cond
     ((empty-p playlist) *empty-playlist-song*)
     ((at-end-p playlist) *end-of-playlist-song*)
     (t (row->song (nth-row (current-idx playlist) (songs-table playlist)))))))

(defun row->song (song-db-entry)
  (with-column-values (file song artist album id3-size) song-db-entry
    (make-instance
     'song
     :file file
     :title (format nil "~a by ~a from ~a" song artist album)
     :id3-size id3-size)))

(defun empty-p (playlist)
  (zerop (table-size (songs-table playlist))))
```

Now, at last, you can implement the method on `maybe-move-to-next-song` that moves `current-idx` to its next value, based on the playlist's repeat mode, and then calls `update-current-if-necessary`. You don't change `current-idx` when it's already at the end of the playlist because you want it to keep its current value, so it'll point at the next song you add to the playlist. This function must lock the playlist before manipulating it since it's called by the Shoutcast server code, which doesn't do any locking.

```
(defmethod maybe-move-to-next-song (song (playlist playlist))
  (with-playlist-locked (playlist)
    (when (still-current-p song playlist)
      (unless (at-end-p playlist)
        (ecase (repeat playlist)
          (:song) ; nothing changes
          (:none (incf (current-idx playlist)))
          (:all  (setf (current-idx playlist)
                       (mod (1+ (current-idx playlist))
                            (table-size (songs-table playlist)))))))
      (update-current-if-necessary playlist))))
```

Manipulating the Playlist

The rest of the playlist code is functions used by the Web interface to manipulate `playlist` objects, including adding and deleting songs, sorting and shuffling, and setting the repeat mode. As in the helper functions in the previous section, you don't need to worry about locking in these functions because, as you'll see, the lock will be acquired in the Web interface function that calls these.

Adding and deleting is mostly a question of manipulating the `songs-table`. The only extra work you have to do is to keep the `current-song` and `current-idx` in sync. For instance, whenever the playlist is empty, its `current-idx` will be zero, and the `current-song` will be the `*empty-playlist-song*`. If you add a song to an empty playlist, then the index of zero is now in bounds, and you should change the `current-song` to the newly added song. By the same token, when you've played all the songs in a playlist and `current-song` is `*end-of-playlist-song*`, adding a song should cause `current-song` to be reset. All this really means, though, is that you need to call `update-current-if-necessary` at the appropriate points.

Adding songs to a playlist is a bit involved because of the way the Web interface communicates which songs to add. For reasons I'll discuss in the next section, the Web interface code can't just give you a simple set of criteria to use in selecting songs from the database. Instead, it gives you the name of a column and a list of values, and you're supposed to add all the songs from the main database where the given column has a value in the list of values. Thus, to add the right songs, you need to first build a table object containing the

desired values, which you can then use with an `in` query against the song database. So, `add-songs` looks like this:

```
(defun add-songs (playlist column-name values)
  (let ((table (make-instance
                'table
                :schema (extract-schema (list column-name) (schema *mp3s*)))))
    (dolist (v values) (insert-row (list column-name v) table))
    (do-rows (row (select :from *mp3s* :where (in column-name table)))
      (insert-row row (songs-table playlist))))
  (update-current-if-necessary playlist))
```

Deleting songs is a bit simpler; you just need to be able to delete songs from the `songs-table` that match particular criteria--either a particular song or all songs in a particular genre, by a particular artist, or from a particular album. So, you can provide a `delete-songs` function that takes keyword/value pairs, which are used to construct a `matching :where` clause you can pass to the `delete-rows` database function.

Another complication that arises when deleting songs is that `current-idx` may need to change. Assuming the current song isn't one of the ones just deleted, you'd like it to remain the current song. But if songs before it in `songs-table` are deleted, it'll be in a different position in the table after the delete. So after a call to `delete-rows`, you need to look for the row containing the current song and reset `current-idx`. If the current song has itself been deleted, then, for lack of anything better to do, you can reset `current-idx` to zero. After updating `current-idx`, calling `update-current-if-necessary` will take care of updating `current-song`. And if `current-idx` changed but still points at the same song, `current-song` will be left alone.

```
(defun delete-songs (playlist &rest names-and-values)
  (delete-rows
   :from (songs-table playlist)
   :where (apply #'matching (songs-table playlist) names-and-values))
  (setf (current-idx playlist) (or (position-of-current playlist) 0))
  (update-current-if-necessary playlist))
```

```
(defun position-of-current (playlist)
  (let* ((table (songs-table playlist))
         (matcher (matching table :file (file (current-song playlist))))
         (pos 0))
    (do-rows (row table)
      (when (funcall matcher row)
        (return-from position-of-current pos))
      (incf pos))))
```

You can also provide a function to completely clear the playlist, which uses `delete-all-rows` and doesn't have to worry about finding the current song since it has obviously been deleted. The call to `update-current-if-necessary` will take care of setting `current-song` to **`NIL`**.

```
(defun clear-playlist (playlist)
  (delete-all-rows (songs-table playlist))
  (setf (current-idx playlist) 0)
  (update-current-if-necessary playlist))
```

Sorting and shuffling the playlist are related in that the playlist is always either sorted *or* shuffled. The `shuffle` slot says whether the playlist should be shuffled and if so how. If it's set to `:none`, then the playlist is ordered according to the value in the `ordering` slot. When `shuffle` is `:song`, the playlist will be randomly permuted. And when it's set to `:album`, the list of albums is randomly permuted, but the songs within each album are listed in track order. Thus, the `sort-playlist` function, which will be called by the Web interface code whenever the user selects a new ordering, needs to set `ordering` to the desired ordering and set `shuffle` to `:none` before calling `order-playlist`, which actually does the sort. As in `delete-songs`, you need to use `position-of-current` to reset `current-idx` to the new location of the current song. However, this time you don't need to call `update-current-if-necessary` since you know the current song is still in the table.

```
(defun sort-playlist (playlist ordering)
  (setf (ordering playlist) ordering)
  (setf (shuffle playlist) :none)
  (order-playlist playlist)
  (setf (current-idx playlist) (position-of-current playlist)))
```

In `order-playlist`, you can use the database function `sort-rows` to actually perform the sort, passing a list of columns to sort by based on the value of `ordering`.

```
(defun order-playlist (playlist)
  (apply #'sort-rows (songs-table playlist)
    (case (ordering playlist)
      (:genre  '(:genre :album :track))
      (:artist '(:artist :album :track))
      (:album  '(:album :track))
      (:song   '(:song)))))
```

The function `shuffle-playlist`, called by the Web interface code when the user selects a new shuffle mode, works in a similar fashion except it doesn't need to change the value of `ordering`. Thus, when `shuffle-playlist` is called with a `shuffle` of `:none`, the playlist goes back to being sorted according to the most recent ordering. Shuffling by songs is simple--just call `shuffle-table` on `songs-table`. Shuffling by albums is a bit more involved but still not rocket science.

```
(defun shuffle-playlist (playlist shuffle)
  (setf (shuffle playlist) shuffle)
  (case shuffle
    (:none (order-playlist playlist))
    (:song (shuffle-by-song playlist))
    (:album (shuffle-by-album playlist)))
  (setf (current-idx playlist) (position-of-current playlist)))

(defun shuffle-by-song (playlist)
  (shuffle-table (songs-table playlist)))

(defun shuffle-by-album (playlist)
  (let ((new-table (make-playlist-table)))
    (do-rows (album-row (shuffled-album-names playlist))
      (do-rows (song (songs-for-album playlist (column-value album-row :album)))
        (insert-row song new-table)))
    (setf (songs-table playlist) new-table)))

(defun shuffled-album-names (playlist)
  (shuffle-table
   (select
    :columns :album
    :from (songs-table playlist)
```

```
    :distinct t)))

(defun songs-for-album (playlist album)
  (select
   :from (songs-table playlist)
   :where (matching (songs-table playlist) :album album)
   :order-by :track))
```

The last manipulation you need to support is setting the playlist's repeat mode. Most of the time you don't need to take any extra action when setting `repeat`--its value comes into play only in `maybe-move-to-next-song`. However, you need to update the `current-song` as a result of changing `repeat` in one situation, namely, if `current-idx` is at the end of a nonempty playlist and `repeat` is being changed to `:song` or `:all`. In that case, you want to continue playing, either repeating the last song or starting at the beginning of the playlist. So, you should define an `:after` method on the generic function `(setf repeat)`.

```
(defmethod (setf repeat) :after (value (playlist playlist))
  (if (and (at-end-p playlist) (not (empty-p playlist)))
    (ecase value
      (:song (setf (current-idx playlist) (1- (table-size (songs-table playlist)))))
      (:none)
      (:all  (setf (current-idx playlist) 0)))
    (update-current-if-necessary playlist)))
```

Now you have all the underlying bits you need. All that remains is the code that will provide a Web-based user interface for browsing the MP3 database and manipulating playlists. The interface will consist of three main functions defined with `define-url-function`: one for browsing the song database, one for viewing and manipulating a single playlist, and one for listing all the available playlists.

But before you get to writing these three functions, you need to start with some helper functions and HTML macros that they'll use.

Query Parameter Types

Since you'll be using `define-url-function`, you need to define a few methods on the `string->type` generic function from Chapter 28 that `define-url-function` uses to convert string query parameters into Lisp objects. In this application, you'll need methods to convert strings to integers, keyword symbols, and a list of values.

The first two are quite simple.

```
(defmethod string->type ((type (eql 'integer)) value)
  (parse-integer (or value "") :junk-allowed t))

(defmethod string->type ((type (eql 'keyword)) value)
  (and (plusp (length value)) (intern (string-upcase value) :keyword)))
```

The last `string->type` method is slightly more complex. For reasons I'll get to in a moment, you'll need to generate pages that display a form that contains a hidden field whose value is a list of strings. Since you're responsible for generating the value in the hidden field *and* for parsing it when it comes back, you can use whatever encoding is convenient. You could use the functions **`WRITE-TO-STRING`** and **`READ-FROM-STRING`**, which use the Lisp printer and reader to write and read data to and from strings, except the printed representation of strings can contain quotation marks and other characters that may cause problems when embedded in the value attribute of an `INPUT` element. So, you'll need to escape those characters somehow. Rather than trying to come up with your own escaping scheme, you can just use base 64, an encoding commonly used to protect binary data sent through e-mail. AllegroServe comes with two functions, `base64-encode` and `base64-decode`, that do the encoding and decoding for you, so all you have to do is write a pair of functions:

one that encodes a Lisp object by converting it to a readable string with **WRITE-TO-STRING** and then base 64 encoding it and, conversely, another to decode such a string by base 64 decoding it and passing the result to **READ-FROM-STRING**. You'll want to wrap the calls to **WRITE-TO-STRING** and **READ-FROM-STRING** in **WITH-STANDARD-IO-SYNTAX** to make sure all the variables that affect the printer and reader are set to their standard values. However, because you're going to be reading data that's coming in from the network, you'll definitely want to turn off one feature of the reader--the ability to evaluate arbitrary Lisp code while reading![4] You can define your own macro `with-safe-io-syntax`, which wraps its body forms in **WITH-STANDARD-IO-SYNTAX** wrapped around a **LET** that binds ***READ-EVAL*** to **NIL**.

```
(defmacro with-safe-io-syntax (&body body)
  `(with-standard-io-syntax
     (let ((*read-eval* nil))
       ,@body)))
```

Then the encoding and decoding functions are trivial.

```
(defun obj->base64 (obj)
  (base64-encode (with-safe-io-syntax (write-to-string obj))))

(defun base64->obj (string)
  (ignore-errors
    (with-safe-io-syntax (read-from-string (base64-decode string)))))
```

Finally, you can use these functions to define a method on `string->type` that defines the conversion for the query parameter type `base64-list`.

```
(defmethod string->type ((type (eql 'base-64-list)) value)
  (let ((obj (base64->obj value)))
    (if (listp obj) obj nil)))
```

Boilerplate HTML

Next you need to define some HTML macros and helper functions to make it easy to give the different pages in the application a consistent look and feel. You can start with an HTML macro that defines the basic structure of a page in the application.

```
(define-html-macro :mp3-browser-page ((&key title (header title)) &body body)
  `(:html
     (:head
      (:title ,title)
      (:link :rel "stylesheet" :type "text/css" :href "mp3-browser.css"))
     (:body
      (standard-header)
      (when ,header (html (:h1 :class "title" ,header)))
      ,@body
      (standard-footer))))
```

You should define `standard-header` and `standard-footer` as separate functions for two reasons. First, during development you can redefine those functions and see the effect immediately without having to recompile functions that use the `:mp3-browser-page` macro. Second, it turns out that one of the pages you'll write later won't be defined with `:mp3-browser-page` but will still need the standard header and footers. They look like this:

```
(defparameter *r* 25)

(defun standard-header ()
  (html
   ((:p :class "toolbar")
    "[" (:a :href (link "/browse" :what "genre") "All genres") "] "
    "[" (:a :href (link "/browse" :what "genre" :random *r*) "Random genres") "] "
    "[" (:a :href (link "/browse" :what "artist") "All artists") "] "
    "[" (:a :href (link "/browse" :what "artist" :random *r*) "Random artists") "] "
    "[" (:a :href (link "/browse" :what "album") "All albums") "] "
    "[" (:a :href (link "/browse" :what "album" :random *r*) "Random albums") "] "
    "[" (:a :href (link "/browse" :what "song" :random *r*) "Random songs") "] "
    "[" (:a :href (link "/playlist") "Playlist") "] "
    "[" (:a :href (link "/all-playlists") "All playlists") "]")))

(defun standard-footer ()
  (html (:hr) ((:p :class "footer") "MP3 Browser v" *major-version* "." *minor-version*)))
```

A couple of smaller HTML macros and helper functions automate other common patterns. The `:table-row` HTML macro makes it easier to generate the HTML for a single row of a table. It uses a feature of FOO

that I'll discuss in Chapter 31, an `&attributes` parameter, which causes uses of the macro to be parsed just like normal s-expression HTML forms, with any attributes gathered into a list that will be bound to the `&attributes` parameter. It looks like this:

```
(define-html-macro :table-row (&attributes attrs &rest values)
  `(:tr ,@attrs ,@(loop for v in values collect `(:td ,v))))
```

And the `link` function generates a URL back into the application to be used as the `HREF` attribute with an `A` element, building a query string out of a set of keyword/value pairs and making sure all special characters are properly escaped. For instance, instead of writing this:

```
(:a :href "browse?what=artist&genre=Rhythm+%26+Blues" "Artists")
```

you can write the following:

```
(:a :href (link "browse" :what "artist" :genre "Rhythm & Blues") "Artists")
```

It looks like this:

```
(defun link (target &rest attributes)
  (html
    (:attribute
     (:format "~a~@[?~{~(~a~)=~a~^&~}~]" target (mapcar #'urlencode attributes)))))
```

To URL encode the keys and values, you use the helper function `urlencode`, which is a wrapper around the function `encode-form-urlencoded`, which is a nonpublic function from AllegroServe. This is--on one hand--bad form; since the name `encode-form-urlencoded` isn't exported from `NET.ASERVE`, it's possible that `encode-form-urlencoded` may go away or get renamed out from under you. On the other hand, using this unexported symbol for the time being lets you get work done for the moment; by wrapping `encode-form-urlencoded` in your own function, you isolate the crufty code to one function, which you could rewrite if you had to.

```
(defun urlencode (string)
  (net.aserve::encode-form-urlencoded string))
```

Finally, you need the CSS style sheet `mp3-browser.css` used by `:mp3-browser-page`. Since there's nothing dynamic about it, it's probably easiest to just publish a static file with `publish-file`.

```
(publish-file :path "/mp3-browser.css"  :file filename :content-type "text/css")
```

A sample style sheet is included with the source code for this chapter on the book's Web site. You'll define a function, at the end of this chapter, that starts the MP3 browser application. It'll take care of, among other things, publishing this file.

The Browse Page

The first URL function will generate a page for browsing the MP3 database. Its query parameters will tell it what kind of thing the user is browsing and provide the criteria of what elements of the database they're interested in. It'll give them a way to select database entries that match a specific genre, artist, or album. In the interest of serendipity, you can also provide a way to select a random subset of matching items. When the user is browsing at the level of individual songs, the title of the song will be a link that causes that song to be added to the playlist. Otherwise, each item will be presented with links that let the user browse the listed item by some other category. For example, if the user is browsing genres, the entry "Blues" will contain links to browse all albums, artists, and songs in the genre Blues. Additionally, the browse page will feature an "Add all" button that adds every song matching the page's criteria to the user's playlist. The function looks like this:

```
(define-url-function browse
    (request (what keyword :genre) genre artist album (random integer))

  (let* ((values (values-for-page what genre artist album random))
         (title (browse-page-title what random genre artist album))
         (single-column (if (eql what :song) :file what))
         (values-string (values->base-64 single-column values)))
    (html
     (:mp3-browser-page
      (:title title)
      ((:form :method "POST" :action "playlist")
       (:input :name "values" :type "hidden" :value values-string)
       (:input :name "what" :type "hidden" :value single-column)
       (:input :name "action" :type "hidden" :value :add-songs)
       (:input :name "submit" :type "submit" :value "Add all"))
      (:ul (do-rows (row values) (list-item-for-page what row)))))))
```

This function starts by using the function `values-for-page` to get a table containing the values it needs to present. When the user is browsing by song--when the `what` parameter is `:song`--you want to select complete rows from the database. But when they're browsing by genre, artist, or album, you want to select only the distinct values for the given category.

The database function `select` does most of the heavy lifting, with `values-for-page` mostly responsible for passing the right arguments depending on the value of `what`. This is also where you select a random subset of the matching rows if necessary.

```
(defun values-for-page (what genre artist album random)
  (let ((values
         (select
          :from *mp3s*
          :columns (if (eql what :song) t what)
          :where (matching *mp3s* :genre genre :artist artist :album album)
          :distinct (not (eql what :song))
          :order-by (if (eql what :song) '(:album :track) what))))
    (if random (random-selection values random) values)))
```

To generate the title for the browse page, you pass the browsing criteria to the following function, `browse-page-title`:

```
(defun browse-page-title (what random genre artist album)
  (with-output-to-string (s)
    (when random (format s "~:(~r~) Random " random))
    (format s "~:(~a~p~)" what random)
    (when (or genre artist album)
      (when (not (eql what :song)) (princ " with songs" s))
      (when genre  (format s " in genre ~a" genre))
      (when artist (format s " by artist ~a " artist))
      (when album  (format s " on album ~a" album)))))
```

Once you have the values you want to present, you need to do two things with them. The main task, of course, is to present them, which happens in the `do-rows` loop, leaving the rendering of each row to the function `list-item-for-page`. That function renders `:song` rows one way and all other kinds another way.

```
(defun list-item-for-page (what row)
  (if (eql what :song)
    (with-column-values (song file album artist genre) row
      (html
        (:li
         (:a :href (link "playlist" :file file :action "add-songs") (:b song)) " from "
         (:a :href (link "browse"  :what :song :album  album) album) " by "
         (:a :href (link "browse" :what :song :artist artist) artist) " in genre "
         (:a :href (link "browse"  :what :song :genre  genre) genre))))
    (let ((value (column-value row what)))
      (html
       (:li value " - "
            (browse-link :genre  what value)
            (browse-link :artist what value)
            (browse-link :album  what value)
            (browse-link :song   what value))))))
```

```
(defun browse-link (new-what what value)
  (unless (eql new-what what)
    (html
     "["
     (:a :href (link "browse" :what new-what what value) (:format "~(~as~)" new-what))
     "] ")))
```

The other thing on the `browse` page is a form with several hidden `INPUT` fields and an "Add all" submit button. You need to use an HTML form instead of a regular link to keep the application stateless--to make sure all the information needed to respond to a request comes in the request itself. Because the browse page results can be partially random, you need to submit a fair bit of data for the server to be able to reconstitute the list of songs to add to the playlist. If you didn't allow the browse page to return randomly generated results, you wouldn't need much data--you could just submit a request to add songs with whatever search criteria the browse page used. But if you added songs that way, with criteria that included a `random` argument, then you'd end up adding a different set of random songs than the user was looking at on the page when they hit the "Add all" button.

The solution you'll use is to send back a form that has enough information stashed away in a hidden `INPUT` element to allow the server to reconstitute the list of songs matching the browse page criteria. That information is the list of values returned by `values-for-page` and the value of the `what` parameter. This is where you use the `base64-list` parameter type; the function `values->base64` extracts the values of a specified column from the table returned by `values-for-page` into a list and then makes a base 64-encoded string out of that list to embed in the form.

```
(defun values->base-64 (column values-table)
  (flet ((value (r) (column-value r column)))
    (obj->base64 (map-rows #'value values-table))))
```

When that parameter comes back as the value of the `values` query parameter to a URL function that declares `values` to be of type `base-64-list`, it'll be automatically converted back to a list. As you'll see in a moment, that list can then be used to construct a query that'll return the correct list of songs.[5] When you're browsing by `:song`, you use the values from the `:file` column since they uniquely identify the actual songs while the song names may not.

The Playlist

This brings me to the next URL function, `playlist`. This is the most complex page of the three--it's responsible for displaying the current contents of the user's playlist as well as for providing the interface to manipulate the playlist. But with most of the tedious bookkeeping handled by `define-url-function`, it's not too hard to see how `playlist` works. Here's the beginning of the definition, with just the parameter list:

```
(define-url-function playlist
    (request
     (playlist-id string (playlist-id request) :package)
     (action keyword)      ; Playlist manipulation action
     (what keyword :file)  ; for :add-songs action
     (values base-64-list) ;             "
     file                  ; for :add-songs and :delete-songs actions
     genre                 ; for :delete-songs action
     artist                ;             "
     album                 ;             "
     (order-by keyword)    ; for :sort action
     (shuffle keyword)     ; for :shuffle action
     (repeat keyword))     ; for :set-repeat action
```

In addition to the obligatory `request` parameter, `playlist` takes a number of query parameters. The most important in some ways is `playlist-id`, which identifies which `playlist` object the page should display and manipulate. For this parameter, you can take advantage of `define-url-function`'s "sticky parameter" feature. Normally, the `playlist-id` won't be supplied explicitly, defaulting to the value returned by the `playlist-id` function, namely, the IP address of the client machine on which the browser is running. However, users can also manipulate their playlists from different machines than the ones running their MP3 clients by allowing this value to be explicitly specified. And if it's specified once, `define-url-function` will arrange for it to "stick" by setting a cookie in the browser. Later you'll define a

URL function that generates a list of all existing playlists, which users can use to pick a playlist other than the one for the machines they're browsing from.

The `action` parameter specifies some action to take on the user's playlist object. The value of this parameter, which will be converted to a keyword symbol for you, can be `:add-songs`, `:delete-songs`, `:clear`, `:sort`, `:shuffle`, or `:set-repeat`. The `:add-songs` action is used by the "Add all" button in the browse page and also by the links used to add individual songs. The other actions are used by the links on the playlist page itself.

The `file`, `what`, and `values` parameters are used with the `:add-songs` action. By declaring `values` to be of type `base-64-list`, the `define-url-function` infrastructure will take care of decoding the value submitted by the "Add all" form. The other parameters are used with other actions as noted in the comments.

Now let's look at the body of `playlist`. The first thing you need to do is use the `playlist-id` to look up the queue object and then acquire the playlist's lock with the following two lines:

```
(let ((playlist (lookup-playlist playlist-id)))
  (with-playlist-locked (playlist)
```

Since `lookup-playlist` will create a new playlist if necessary, this will always return a `playlist` object. Then you take care of any necessary queue manipulation, dispatching on the value of the `action` parameter in order to call one of the `playlist` functions.

```
(case action
  (:add-songs      (add-songs playlist what (or values (list file))))
  (:delete-songs   (delete-songs
                    playlist
                    :file file :genre genre
```

```
                    :artist artist :album album))
  (:clear           (clear-playlist playlist))
  (:sort            (sort-playlist playlist order-by))
  (:shuffle         (shuffle-playlist playlist shuffle))
  (:set-repeat      (setf (repeat playlist) repeat)))
```

All that's left of the `playlist` function is the actual HTML generation. Again, you can use the `:mp3-browser-page` HTML macro to make sure the basic form of the page matches the other pages in the application, though this time you pass **`NIL`** to the `:header` argument in order to leave out the `H1` header. Here's the rest of the function:

```
(html
 (:mp3-browser-page
  (:title (:format "Playlist - ~a" (id playlist)) :header nil)
  (playlist-toolbar playlist)
  (if (empty-p playlist)
    (html (:p (:i "Empty.")))
    (html
      ((:table :class "playlist")
       (:table-row "#" "Song" "Album" "Artist" "Genre")
       (let ((idx 0)
             (current-idx (current-idx playlist)))
         (do-rows (row (songs-table playlist))
           (with-column-values (track file song album artist genre) row
             (let ((row-style (if (= idx current-idx) "now-playing" "normal")))
               (html
                 ((:table-row :class row-style)
                  track
                  (:progn song   (delete-songs-link :file file))
                  (:progn album  (delete-songs-link :album album))
                  (:progn artist (delete-songs-link :artist artist))
                  (:progn genre  (delete-songs-link :genre genre)))))
             (incf idx)))))))))))
```

The function `playlist-toolbar` generates a toolbar containing links to `playlist` to perform the various `:action` manipulations. And `delete-songs-link` generates a link to `playlist` with the `:action` parameter set to `:delete-songs` and the appropriate arguments to delete an individual file, or all files on an album, by a particular artist or in a specific genre.

```
(defun playlist-toolbar (playlist)
  (let ((current-repeat (repeat playlist))
        (current-sort (ordering playlist))
        (current-shuffle (shuffle playlist)))
    (html
     (:p :class "playlist-toolbar"
```

```
          (:i "Sort by:")
          " [ "
          (sort-playlist-button "genre" current-sort) " | "
          (sort-playlist-button "artist" current-sort) " | "
          (sort-playlist-button "album" current-sort) " | "
          (sort-playlist-button "song" current-sort) " ] "
          (:i "Shuffle by:")
          " [ "
          (playlist-shuffle-button "none" current-shuffle) " | "
          (playlist-shuffle-button "song" current-shuffle) " | "
          (playlist-shuffle-button "album" current-shuffle) " ] "
          (:i "Repeat:")
          " [ "
          (playlist-repeat-button "none" current-repeat) " | "
          (playlist-repeat-button "song" current-repeat) " | "
          (playlist-repeat-button "all" current-repeat) " ] "
          "[ " (:a :href (link "playlist" :action "clear") "Clear") " ] "))))

(defun playlist-button (action argument new-value current-value)
  (let ((label (string-capitalize new-value)))
    (if (string-equal new-value current-value)
      (html (:b label))
      (html (:a :href (link "playlist" :action action argument new-value) label)))))

(defun sort-playlist-button (order-by current-sort)
  (playlist-button :sort :order-by order-by current-sort))

(defun playlist-shuffle-button (shuffle current-shuffle)
  (playlist-button :shuffle :shuffle shuffle current-shuffle))

(defun playlist-repeat-button (repeat current-repeat)
  (playlist-button :set-repeat :repeat repeat current-repeat))

(defun delete-songs-link (what value)
  (html " [" (:a :href (link "playlist" :action :delete-songs what value) "x") "]"))
```

Finding a Playlist

The last of the three URL functions is the simplest. It presents a table listing all the playlists that have been created. Ordinarily users won't need to use this page, but during development it gives you a useful view into the state of the system. It also provides the mechanism to choose a different playlist--each playlist ID is a link to the `playlist` page with an explicit `playlist-id` query parameter, which will then be made sticky by the `playlist` URL function. Note that you need to acquire the `*playlists-lock*` to make sure the `*playlists*` hash table doesn't change out from under you while you're iterating over it.

```
(define-url-function all-playlists (request)
  (:mp3-browser-page
   (:title "All Playlists")
   ((:table :class "all-playlists")
    (:table-row "Playlist" "# Songs" "Most recent user agent")
    (with-process-lock (*playlists-lock*)
      (loop for playlist being the hash-values of *playlists* do
           (html
             (:table-row
              (:a :href (link "playlist" :playlist-id (id playlist)) (:print (id playlist)))
              (:print (table-size (songs-table playlist)))
              (:print (user-agent playlist)))))))))
```

Running the App

And that's it. To use this app, you just need to load the MP3 database with the `load-database` function from Chapter 27, publish the CSS style sheet, set `*song-source-type*` to `playlist` so `find-song-source` uses playlists instead of the singleton song source defined in the previous chapter, and start AllegroServe. The following function takes care of all these steps for you, after you fill in appropriate values for the two parameters `*mp3-dir*`, which is the root directory of your MP3 collection, and `*mp3-css*`, the filename of the CSS style sheet:

```
(defparameter *mp3-dir* ...)

(defparameter *mp3-css* ...)

(defun start-mp3-browser ()
  (load-database *mp3-dir* *mp3s*)
  (publish-file :path "/mp3-browser.css"  :file *mp3-css* :content-type "text/css")
  (setf *song-source-type* 'playlist)
  (net.aserve::debug-on :notrap)
  (net.aserve:start :port 2001))
```

When you invoke this function, it will print dots while it loads the ID3 information from your ID3 files. Then you can point your MP3 client at this URL:

```
http://localhost:2001/stream.mp3
```

and point your browser at some good starting place, such as this:

```
http://localhost:2001/browse
```

which will let you start browsing by the default category, Genre. After you've added some songs to the playlist, you can press Play on the MP3 client, and it should start playing the first song.

Obviously, you could improve the user interface in any of a number of ways--for instance, if you have a lot of MP3s in your library, it might be useful to be able to browse artists or albums by the first letter of their names. Or maybe you could add a "Play whole album" button to the playlist page that causes the playlist to immediately put all the songs from the same album as the currently playing song at the top of the playlist. Or you could change the `playlist` class, so instead of playing silence when there are no songs queued up, it picks a random song from the database. But all those ideas fall in the realm of application design, which isn't really the topic of this book. Instead, the next two chapters will drop back to the level of software infrastructure to cover how the FOO HTML generation library works.

---

[1]The intricacies of concurrent programming are beyond the scope of this book. The basic idea is that if you have multiple threads of control--as you will in this application with some threads running the `shoutcast` function and other threads responding to requests from the browser--then you need to make sure only one thread at a time manipulates an object in order to prevent one thread from seeing the object in an inconsistent state while another thread is working on it. In this function, for instance, if two new MP3 clients are connecting at the same time, they'd both try to add an entry to `*playlists*` and might interfere with each other. The `with-process-lock` ensures that each thread gets exclusive access to the hash table for long enough to do the work it needs to do.

[2]This approach also assumes that every client machine has a unique IP address. This assumption should hold as long as all the users are on the same LAN but may not hold if clients are connecting from behind a firewall that does network address translation. Deploying this application outside a LAN will require some modifications, but if you want to deploy this application to the wider Internet, you'd better know enough about networking to figure out an appropriate scheme yourself.

[3]Unfortunately, because of licensing issues around the MP3 format, it's not clear that it's legal for me to provide you with such an MP3 without paying licensing fees to Fraunhofer IIS. I got mine as part of the software that came with my Slimp3 from Slim Devices. You can grab it from their Subversion repository via the Web at

`http://svn.slimdevices.com/*checkout*/trunk/server/ HTML/EN/html/silentpacket.mp3?rev=2`. Or buy a Squeezebox, the new, wireless version of Slimp3, and you'll get `silentpacket.mp3` as part of the software that comes with it. Or find an MP3 of John Cage's piece *4'33"*.

[4]The reader supports a bit of syntax, `#.`, that causes the following s-expression to be evaluated at read time. This is occasionally useful in source code but obviously opens a big security hole when you read untrusted data. However, you can turn off this syntax by setting **`*READ-EVAL*`** to **`NIL`**, which will cause the reader to signal an error if it encounters `#.`.

[5]This solution has its drawbacks--if a `browse` page returns a lot of results, a fair bit of data is going back and forth under the covers. Also, the database queries aren't necessarily the most efficient. But it does keep the application stateless. An alternative approach is to squirrel away, on the server side, information about the results returned by `browse` and then, when a request to add songs come in, find the appropriate bit of information in order to re-create the correct set of songs. For instance, you could just save the values list instead of sending it back in the form. Or you could copy the **`RANDOM-STATE`** object before you generate the browse results so you can later re-create the same "random" results. But this approach causes its own problems. For instance, you'd then need to worry about when you can get rid of the squirreled-away information; you never know when the user might hit the Back button on their browser to return to an old browse page and then hit the "Add all" button. Welcome to the wonderful world of Web programming.

Copyright © 2003-2005, Peter Seibel

30. Practical: An HTML Generation Library, the Interpreter

In this chapter and the next you'll take a look under the hood of the FOO HTML generator that you've been using in the past few chapters. FOO is an example of a kind of programming that's quite common in Common Lisp and relatively uncommon in non-Lisp languages, namely, *language-oriented* programming. Rather than provide an API built primarily out of functions, classes, and macros, FOO provides language processors for a domain-specific language that you can embed in your Common Lisp programs.

FOO provides two language processors for the same s-expression language. One is an interpreter that takes a FOO "program" as data and interprets it to generate HTML. The other is a compiler that compiles FOO expressions, possibly with embedded Common Lisp code, into Common Lisp that generates HTML and runs the embedded code. The interpreter is exposed as the function `emit-html` and the compiler as the macro `html`, which you used in previous chapters.

In this chapter you'll look at some of the infrastructure shared between the interpreter and the compiler and then at the implementation of the interpreter. In the next chapter, I'll show you how the compiler works.

Designing a Domain-Specific Language

Designing an embedded language requires two steps: first, design the language that'll allow you to express the things you want to express, and second, implement a processor, or processors, that accepts a "program" in that language and either performs the actions indicated by the program or translates the program into Common Lisp code that'll perform equivalent behaviors.

So, step one is to design the HTML-generating language. The key to designing a good domain-specific language is to strike the right balance between expressiveness and concision. For instance, a highly expressive but not very concise "language" for generating HTML is the language of literal HTML strings. The legal "forms" of this language are strings containing literal HTML. Language processors for this "language" could process such forms by simply emitting them as-is.

```
(defvar *html-output* *standard-output*)

(defun emit-html (html)
  "An interpreter for the literal HTML language."
  (write-sequence html *html-output*))

(defmacro html (html)
  "A compiler for the literal HTML language."
  `(write-sequence ,html *html-output*))
```

This "language" is highly expressive since it can express *any* HTML you could possibly want to generate.[1] On the other hand, this language doesn't win a lot of points for its concision because it gives you zero compression--its input *is* its output.

To design a language that gives you some useful compression without sacrificing too much expressiveness, you need to identify the details of the output that are either redundant or uninteresting. You can then make those aspects of the output implicit in the semantics of the language.

For instance, because of the structure of HTML, every opening tag is paired with a matching closing tag.[2] When you write HTML by hand, you have to write those closing tags, but you can improve the concision of your HTML-generating language by making the closing tags implicit.

Another way you can gain concision at a slight cost in expressiveness is to make the language processors responsible for adding appropriate whitespace between elements--blank lines and indentation. When you're generating HTML programmatically, you typically don't care much about which elements have line breaks before or after them or about whether different elements are indented relative to their parent elements. Letting the language processor insert whitespace according to some rule means you don't have to worry about it. As it turns out, FOO actually supports two modes--one that uses the minimum amount of whitespace, which allows it to generate extremely efficient code and compact HTML, and another that generates nicely formatted HTML with different elements indented and separated from other elements according to their role.

Another detail that's best moved into the language processor is the escaping of certain characters that have a special meaning in HTML

such as `<`, `>`, and `&`. Obviously, if you generate HTML by just printing strings to a stream, then it's up to you to replace any occurrences of those characters in the string with the appropriate escape sequences, `<`, `>` and `&`. But if the language processor can know which strings are to be emitted as element data, then it can take care of automatically escaping those characters for you.

The FOO Language

So, enough theory. I'll give you a quick overview of the language implemented by FOO, and then you'll look at the implementation of the two FOO language processors--the interpreter, in this chapter, and the compiler, in the next.

Like Lisp itself, the basic syntax of the FOO language is defined in terms of forms made up of Lisp objects. The language defines how each legal FOO form is translated into HTML.

The simplest FOO forms are self-evaluating Lisp objects such as strings, numbers, and keyword symbols.[3] You'll need a function `self-evaluating-p` that tests whether a given object is self-evaluating for FOO's purposes.

```
(defun self-evaluating-p (form)
  (and (atom form) (if (symbolp form) (keywordp form) t)))
```

Objects that satisfy this predicate will be emitted by converting them to strings with **PRINC-TO-STRING** and then escaping any reserved characters, such as `<`, `>`, or `&`. When the value is being emitted as an attribute, the characters `"`, and `'` are also escaped. Thus, you can invoke the `html` macro on a self-evaluating object to emit it to `*html-output*` (which is initially bound to ***STANDARD-OUTPUT***). Table 30-1 shows how a few different self-evaluating values will be output.

Table 30-1. FOO Output for Self-Evaluating Objects

FOO Form	Generated HTML
`"foo"`	`foo`
`10`	`10`
`:foo`	`FOO`

```
"foo & bar"                 foo &amp; bar
```

Of course, most HTML consists of tagged elements. The three pieces of information that describe each element are the tag, a set of attributes, and a body containing text and/or more HTML elements. Thus, you need a way to represent these three pieces of information as Lisp objects, preferably ones that the Lisp reader already knows how to read.[4] If you forget about attributes for a moment, there's an obvious mapping between Lisp lists and HTML elements: any HTML element can be represented by a list whose **FIRST** is a symbol where the name is the name of the element's tag and whose **REST** is a list of self-evaluating objects or lists representing other HTML elements. Thus:

```
<p>Foo</p> <==> (:p "Foo")

<p><i>Now</i> is the time</p> <==> (:p (:i "Now") " is the time")
```

Now the only problem is where to squeeze in the attributes. Since most elements have no attributes, it'd be nice if you could use the preceding syntax for elements without attributes. FOO provides two ways to notate elements with attributes. The first is to simply include the attributes in the list immediately following the symbol, alternating keyword symbols naming the attributes and objects representing the attribute value forms. The body of the element starts with the first item in the list that's in a position to be an attribute name and isn't a keyword symbol. Thus:

```
HTML> (html (:p "foo"))
<p>foo</p>
NIL
HTML> (html (:p "foo " (:i "bar") " baz"))
<p>foo <i>bar</i> baz</p>
NIL
HTML> (html (:p :style "foo" "Foo"))
<p style='foo'>Foo</p>
```

```
NIL
HTML> (html (:p :id "x" :style "foo" "Foo"))
<p id='x' style='foo'>Foo</p>
NIL
```

For folks who prefer a bit more obvious delineation between the element's attributes and its body, FOO supports an alternative syntax: if the first element of a list is itself a list with a keyword as *its* first element, then the outer list represents an HTML element with that keyword indicating the tag, with the **REST** of the nested list as the attributes, and with the **REST** of the outer list as the body. Thus, you could write the previous two expressions like this:

```
HTML> (html ((:p :style "foo") "Foo"))
<p style='foo'>Foo</p>
NIL
HTML> (html ((:p :id "x" :style "foo") "Foo"))
<p id='x' style='foo'>Foo</p>
NIL
```

The following function tests whether a given object matches either of these syntaxes:

```
(defun cons-form-p (form &optional (test #'keywordp))
  (and (consp form)
       (or (funcall test (car form))
           (and (consp (car form)) (funcall test (caar form))))))
```

You should parameterize the `test` function because later you'll need to test the same two syntaxes with a slightly different predicate on the name.

To completely abstract the differences between the two syntax variants, you can define a function, `parse-cons-form`, that takes a form and parses it into three elements, the tag, the attributes plist, and the body list, returning them as multiple values. The code that actually evaluates cons forms will use this function and not have to worry about which syntax was used.

```
(defun parse-cons-form (sexp)
  (if (consp (first sexp))
    (parse-explicit-attributes-sexp sexp)
    (parse-implicit-attributes-sexp sexp)))

(defun parse-explicit-attributes-sexp (sexp)
  (destructuring-bind ((tag &rest attributes) &body body) sexp
    (values tag attributes body)))

(defun parse-implicit-attributes-sexp (sexp)
  (loop with tag = (first sexp)
     for rest on (rest sexp) by #'cddr
     while (and (keywordp (first rest)) (second rest))
     when (second rest)
       collect (first rest) into attributes and
       collect (second rest) into attributes
     end
     finally (return (values tag attributes rest))))
```

Now that you have the basic language specified, you can think about how you're actually going to implement the language processors. How do you get from a series of FOO forms to the desired HTML? As I mentioned previously, you'll be implementing two language processors for FOO: an interpreter that walks a tree of FOO forms and emits the corresponding HTML directly and a compiler that walks a tree and translates it into Common Lisp code that'll emit the same HTML. Both the interpreter and compiler will be built on top of a common foundation of code, which provides support for things such as escaping reserved characters and generating nicely indented output, so it makes sense to start there.

Character Escaping

The first bit of the foundation you'll need to lay is the code that knows how to escape characters with a special meaning in HTML. There are three such characters, and they must not appear in the text of an element or in an attribute value; they are `<`, `>`, and `&`. In element text or attribute values, these characters must be replaced with the *character reference entities* `<`, `>`, and `&`. Similarly, in attribute values, the quotation marks used to delimit the value must be escaped, `'` with `'` and `"` with `"`. Additionally, any character can be represented by a numeric character reference entity consisting of an ampersand, followed by a sharp sign, followed by the numeric code as a base 10 integer, and followed by a semicolon. These numeric escapes are sometimes used to embed non-ASCII characters in HTML.

The Package

Since FOO is a low-level library, the package you develop it in doesn't rely on much external code--just the usual dependency on names from the `COMMON-LISP` package and, almost as usual, on the names of the macro-writing macros from `COM.GIGAMONKEYS.MACRO-UTILITIES`. On the other hand, the package needs to export all the names needed by code that uses FOO. Here's the **`DEFPACKAGE`** from the source that you can download from the book's Web site:

```
(defpackage :com.gigamonkeys.html
  (:use :common-lisp :com.gigamonkeys.macro-utilities)
  (:export :with-html-output
           :in-html-style
           :define-html-macro
           :html
           :emit-html
           :&attributes))
```

The following function accepts a single character and returns a string containing a character reference entity for that character:

```
(defun escape-char (char)
  (case char
    (#\& "&amp;")
    (#\< "&lt;")
    (#\> "&gt;")
    (#\' "&apos;")
    (#\" "&quot;")
    (t (format nil "&#~d;" (char-code char)))))
```

You can use this function as the basis for a function, `escape`, that takes a string and a sequence of characters and returns a copy of the first argument with all occurrences of the characters in the second argument replaced with the corresponding character entity returned by `escape-char`.

```
(defun escape (in to-escape)
  (flet ((needs-escape-p (char) (find char to-escape)))
    (with-output-to-string (out)
      (loop for start = 0 then (1+ pos)
            for pos = (position-if #'needs-escape-p in :start start)
            do (write-sequence in out :start start :end pos)
            when pos do (write-sequence (escape-char (char in pos)) out)
            while pos))))
```

You can also define two parameters: `*element-escapes*`, which contains the characters you need to escape in normal element data, and `*attribute-escapes*`, which contains the set of characters to be escaped in attribute values.

```
(defparameter *element-escapes* "<>&")
(defparameter *attribute-escapes* "<>&\"'")
```

Here are some examples:

```
HTML> (escape "foo & bar" *element-escapes*)
"foo &amp; bar"
HTML> (escape "foo & 'bar'" *element-escapes*)
"foo &amp; 'bar'"
HTML> (escape "foo & 'bar'" *attribute-escapes*)
"foo &amp; &apos;bar&apos;"
```

Finally, you'll need a variable, `*escapes*`, that will be bound to the set of characters that need to be escaped. It's initially set to the value of `*element-escapes*`, but when generating attributes, it will, as you'll see, be rebound to the value of `*attribute-escapes*`.

```
(defvar *escapes* *element-escapes*)
```

Indenting Printer

To handle generating nicely indented output, you can define a class `indenting-printer`, which wraps around an output stream, and functions that use an instance of that class to emit strings to the stream while keeping track of when it's at the beginning of the line. The class looks like this:

```
(defclass indenting-printer ()
  ((out                 :accessor out                 :initarg :out)
   (beginning-of-line-p :accessor beginning-of-line-p :initform t)
   (indentation         :accessor indentation         :initform 0)
   (indenting-p         :accessor indenting-p         :initform t)))
```

The main function that operates on `indenting-printer`s is `emit`, which takes the printer and a string and emits the string to the printer's output stream, keeping track of when it emits a newline so it can reset the `beginning-of-line-p` slot.

```
(defun emit (ip string)
  (loop for start = 0 then (1+ pos)
     for pos = (position #\Newline string :start start)
     do (emit/no-newlines ip string :start start :end pos)
     when pos do (emit-newline ip)
     while pos))
```

To actually emit the string, it uses the function `emit/no-newlines`, which emits any needed indentation, via the helper `indent-if-necessary`, and then writes the string to the stream. This function can also be called directly by other code to emit a string that's known not to contain any newlines.

```
(defun emit/no-newlines (ip string &key (start 0) end)
  (indent-if-necessary ip)
  (write-sequence string (out ip) :start start :end end)
  (unless (zerop (- (or end (length string)) start))
    (setf (beginning-of-line-p ip) nil)))
```

The helper `indent-if-necessary` checks `beginning-of-line-p` and `indenting-p` to determine whether it needs to emit indentation and, if they're both true, emits as many spaces as indicated by the value of `indentation`. Code that uses the `indenting-printer` can control the indentation by manipulating the `indentation` and `indenting-p` slots. Incrementing and decrementing `indentation` changes the number of leading spaces, while setting `indenting-p` to **`NIL`** can temporarily turn off indentation.

```
(defun indent-if-necessary (ip)
  (when (and (beginning-of-line-p ip) (indenting-p ip))
    (loop repeat (indentation ip) do (write-char #\Space (out ip)))
    (setf (beginning-of-line-p ip) nil)))
```

The last two functions in the `indenting-printer` API are `emit-newline` and `emit-freshline`, which are both used to emit a newline character, similar to the `~%` and `~&` **`FORMAT`** directives. That is, the only difference is that `emit-newline` always emits a newline, while `emit-freshline` does so only if `beginning-of-line-p` is false. Thus, multiple calls to `emit-freshline` without any intervening `emit`s won't result in a blank line. This is handy when one piece of code wants to generate some output that should end with a newline while another piece of code wants to generate some output that should start on a newline but you don't want a blank line between the two bits of output.

```
(defun emit-newline (ip)
  (write-char #\Newline (out ip))
  (setf (beginning-of-line-p ip) t))

(defun emit-freshline (ip)
  (unless (beginning-of-line-p ip) (emit-newline ip)))
```

With those preliminaries out of the way, you're ready to get to the guts of the FOO processor.

HTML Processor Interface

Now you're ready to define the interface that'll be used by the FOO language processor to emit HTML. You can define this interface as a set of generic functions because you'll need two implementations--one that actually emits HTML and another that the `html` macro can use to collect a list of actions that need to be performed, which can then be optimized and compiled into code that emits the same output in a more efficient way. I'll call this set of generic functions the *backend interface*. It consists of the following eight generic functions:

```
(defgeneric raw-string (processor string &optional newlines-p))

(defgeneric newline (processor))

(defgeneric freshline (processor))

(defgeneric indent (processor))

(defgeneric unindent (processor))

(defgeneric toggle-indenting (processor))

(defgeneric embed-value (processor value))

(defgeneric embed-code (processor code))
```

While several of these functions have obvious correspondence to `indenting-printer` functions, it's important to understand that these generic functions define the abstract operations that are used by the FOO language processors and won't always be implemented in terms of calls to the `indenting-printer` functions.

That said, perhaps the easiest way to understand the semantics of these abstract operations is to look at the concrete implementations of the methods specialized on `html-pretty-printer`, the class used to generate human-readable HTML.

The Pretty Printer Backend

You can start by defining a class with two slots--one to hold an instance of `indenting-printer` and one to hold the tab width--the number of spaces you want to increase the indentation for each level of nesting of HTML elements.

```
(defclass html-pretty-printer ()
  ((printer   :accessor printer   :initarg :printer)
   (tab-width :accessor tab-width :initarg :tab-width :initform 2)))
```

Now you can implement methods specialized on `html-pretty-printer` on the eight generic functions that make up the backend interface.

The FOO processors use the `raw-string` function to emit strings that don't need character escaping, either because you actually want to emit normally reserved characters or because all reserved characters have already been escaped. Usually `raw-string` is invoked with strings that don't contain newlines, so the default behavior is to use `emit/no-newlines` unless the caller specifies a non-**NIL** `newlines-p` argument.

```
(defmethod raw-string ((pp html-pretty-printer) string &optional newlines-p)
  (if newlines-p
    (emit (printer pp) string)
    (emit/no-newlines (printer pp) string)))
```

The functions `newline`, `freshline`, `indent`, `unindent`, and `toggle-indenting` implement fairly straightforward manipulations of the underlying `indenting-printer`. The only wrinkle is that the HTML pretty printer generates pretty output only when the dynamic variable `*pretty*` is true. When it's **NIL**, you should generate compact HTML with no unnecessary whitespace. So, these methods, with the exception of `newline`, all check `*pretty*` before doing anything:[5]

```
(defmethod newline ((pp html-pretty-printer))
  (emit-newline (printer pp)))

(defmethod freshline ((pp html-pretty-printer))
  (when *pretty* (emit-freshline (printer pp))))
```

```
(defmethod indent ((pp html-pretty-printer))
  (when *pretty*
    (incf (indentation (printer pp)) (tab-width pp))))

(defmethod unindent ((pp html-pretty-printer))
  (when *pretty*
    (decf (indentation (printer pp)) (tab-width pp))))

(defmethod toggle-indenting ((pp html-pretty-printer))
  (when *pretty*
    (with-slots (indenting-p) (printer pp)
      (setf indenting-p (not indenting-p)))))
```

Finally, the functions `embed-value` and `embed-code` are used only by the FOO compiler--`embed-value` is used to generate code that'll emit the value of a Common Lisp expression, while `embed-code` is used to embed a bit of code to be run and its result discarded. In the interpreter, you can't meaningfully evaluate embedded Lisp code, so the methods on these functions always signal an error.

```
(defmethod embed-value ((pp html-pretty-printer) value)
  (error "Can't embed values when  interpreting. Value: ~s" value))

(defmethod embed-code ((pp html-pretty-printer) code)
  (error "Can't embed code when interpreting. Code: ~s" code))
```

Using Conditions to Have Your Cake and Eat It Too

An alternate approach would be to use **EVAL** to evaluate Lisp expressions in the interpreter. The problem with this approach is that **EVAL** has no access to the lexical environment. Thus, there's no way to make something like this work:

```
(let ((x 10)) (emit-html '(:p x)))
```

when `x` is a lexical variable. The symbol `x` that's passed to `emit-html` at runtime has no particular connection to the lexical variable named with the same symbol. The Lisp compiler arranges for references to `x` in the code to refer to the variable, but after the code is compiled, there's no longer necessarily any association between the name `x` and that variable. This is the main reason that when you think **EVAL** is the solution to your problem, you're probably wrong.

However, if `x` was a dynamic variable, declared with **DEFVAR** or **DEFPARAMETER** (and likely named `*x*` instead of `x`), **EVAL** could get at its value. Thus, it might be useful to allow the FOO interpreter to use **EVAL** in some situations. But it's a bad idea to always use **EVAL**. You can get the best of both worlds by combining the idea of using **EVAL** with the condition system.

First define some error classes that you can signal when `embed-value` and `embed-code` are called in the interpreter.

```
(define-condition embedded-lisp-in-interpreter (error)
  ((form :initarg :form :reader form)))

(define-condition value-in-interpreter (embedded-lisp-in-interpreter) ()
  (:report
   (lambda (c s)
     (format s "Can't embed values when interpreting. Value: ~s" (form c)))))

(define-condition code-in-interpreter (embedded-lisp-in-interpreter) ()
  (:report
   (lambda (c s)
     (format s "Can't embed code when interpreting. Code: ~s" (form c)))))
```

Now you can implement `embed-value` and `embed-code` to signal those errors *and* provide a restart that'll evaluate the form with **`EVAL`**.

```
(defmethod embed-value ((pp html-pretty-printer) value)
  (restart-case (error 'value-in-interpreter :form value)
    (evaluate ()
      :report (lambda (s) (format s "EVAL ~s in null lexical environment." value))
      (raw-string pp (escape (princ-to-string (eval value)) *escapes*) t))))

(defmethod embed-code ((pp html-pretty-printer) code)
  (restart-case (error 'code-in-interpreter :form code)
    (evaluate ()
      :report (lambda (s) (format s "EVAL ~s in null lexical environment." code))
      (eval code))))
```

Now you can do something like this:

```
HTML> (defvar *x* 10)
*X*
HTML> (emit-html '(:p *x*))
```

and you'll get dropped into the debugger with this message:

```
Can't embed values when interpreting. Value: *X*
   [Condition of type VALUE-IN-INTERPRETER]

Restarts:
  0: [EVALUATE] EVAL *X* in null lexical environment.
  1: [ABORT] Abort handling SLIME request.
  2: [ABORT] Abort entirely from this process.
```

If you invoke the `evaluate` restart, `embed-value` will **`EVAL`** `*x*`, get the value `10`, and generate this HTML:

```
<p>10</p>
```

Then, as a convenience, you can provide restart functions--functions that invoke the `evaluate` restart--in certain situations. The `evaluate` restart function unconditionally invokes the restart, while `eval-dynamic-variables` and `eval-code` invoke it only if the form in the condition is a dynamic variable or potential code.

```
(defun evaluate (&optional condition)
  (declare (ignore condition))
  (invoke-restart 'evaluate))
```

```
(defun eval-dynamic-variables (&optional condition)
  (when (and (symbolp (form condition)) (boundp (form condition)))
    (evaluate)))

(defun eval-code (&optional condition)
  (when (consp (form condition))
    (evaluate)))
```

Now you can use **HANDLER-BIND** to set up a handler to automatically invoke the `evaluate` restart for you.

```
HTML> (handler-bind ((value-in-interpreter #'evaluate)) (emit-html '(:p *x*)))
<p>10</p>
T
```

Finally, you can define a macro to provide a nicer syntax for binding handlers for the two kinds of errors.

```
(defmacro with-dynamic-evaluation ((&key values code) &body body)
  `(handler-bind (
       ,@(if values `((value-in-interpreter #'evaluate)))
       ,@(if code `((code-in-interpreter #'evaluate))))
     ,@body))
```

With this macro defined, you can write this:

```
HTML> (with-dynamic-evaluation (:values t) (emit-html '(:p *x*)))
<p>10</p>
T
```

The Basic Evaluation Rule

Now to connect the FOO language to the processor interface, all you need is a function that takes an object and processes it, invoking the appropriate processor functions to generate HTML. For instance, when given a simple form like this:

```
(:p "Foo")
```

this function might execute this sequence of calls on the processor:

```
(freshline processor)
(raw-string processor "<p" nil)
(raw-string processor ">" nil)
(raw-string processor "Foo" nil)
(raw-string processor "</p>" nil)
(freshline processor)
```

For now you can define a simple function that just checks whether a form is, in fact, a legal FOO form and, if it is, hands it off to the function `process-sexp-html` for processing. In the next chapter, you'll add some bells and whistles to this function to allow it to handle macros and special operators. But for now it looks like this:

```
(defun process (processor form)
  (if (sexp-html-p form)
    (process-sexp-html processor form)
    (error "Malformed FOO form: ~s" form)))
```

The function `sexp-html-p` determines whether the given object is a legal FOO expression, either a self-evaluating form or a properly formatted cons.

```
(defun sexp-html-p (form)
  (or (self-evaluating-p form) (cons-form-p form)))
```

Self-evaluating forms are easily handled: just convert to a string with **PRINC-TO-STRING** and escape the characters in the variable `*escapes*`, which, as you'll recall, is initially bound to the value of `*element-escapes*`. Cons forms you pass off to `process-cons-sexp-html`.

```
(defun process-sexp-html (processor form)
  (if (self-evaluating-p form)
    (raw-string processor (escape (princ-to-string form) *escapes*) t)
    (process-cons-sexp-html processor form)))
```

The function `process-cons-sexp-html` is then responsible for emitting the opening tag, any attributes, the body, and the closing tag. The main complication here is that to generate pretty HTML, you need to emit fresh lines and adjust the indentation according to the type of the element being emitted. You can categorize all the elements defined in HTML into one of three categories: block, paragraph, and inline. Block elements--such as `body` and `ul`--are emitted with fresh lines before and after both their opening and closing tags and with their contents indented one level. Paragraph elements--such as `p`, `li`, and `blockquote`--are emitted with a fresh line before the opening tag and after the closing tag. Inline elements are simply emitted in line. The following three parameters list the elements of each type:

```
(defparameter *block-elements*
  '(:body :colgroup :dl :fieldset :form :head :html :map :noscript :object
    :ol :optgroup :pre :script :select :style :table :tbody :tfoot :thead
    :tr :ul))

(defparameter *paragraph-elements*
  '(:area :base :blockquote :br :button :caption :col :dd :div :dt :h1
    :h2 :h3 :h4 :h5 :h6 :hr :input :li :link :meta :option :p :param
    :td :textarea :th :title))

(defparameter *inline-elements*
  '(:a :abbr :acronym :address :b :bdo :big :cite :code :del :dfn :em
    :i :img :ins :kbd :label :legend :q :samp :small :span :strong :sub
    :sup :tt :var))
```

The functions `block-element-p` and `paragraph-element-p` test whether a given tag is a member of the corresponding list.[6]

```
(defun block-element-p (tag) (find tag *block-elements*))

(defun paragraph-element-p (tag) (find tag *paragraph-elements*))
```

Two other categorizations with their own predicates are the elements that are always empty, such as `br` and `hr`, and the three elements, `pre`, `style`, and `script`, in which whitespace is supposed to be preserved. The former are handled specially when generating regular HTML (in other words, not XHTML) since they're not supposed to have a closing tag. And when emitting the three tags in which whitespace is preserved, you can temporarily turn off indentation so the pretty printer doesn't add any spaces that aren't part of the element's actual contents.

```
(defparameter *empty-elements*
  '(:area :base :br :col :hr :img :input :link :meta :param))

(defparameter *preserve-whitespace-elements* '(:pre :script :style))

(defun empty-element-p (tag) (find tag *empty-elements*))

(defun preserve-whitespace-p (tag) (find tag *preserve-whitespace-elements*))
```

The last piece of information you need when generating HTML is whether you're generating XHTML since that affects how you emit empty elements.

```
(defparameter *xhtml* nil)
```

With all that information, you're ready to process a cons FOO form. You use `parse-cons-form` to parse the list into three parts, the tag symbol, a possibly empty plist of attribute key/value pairs, and a possibly empty list of body forms. You then emit the opening tag, the

body, and the closing tag with the helper functions `emit-open-tag`, `emit-element-body`, and `emit-close-tag`.

```
(defun process-cons-sexp-html (processor form)
  (when (string= *escapes* *attribute-escapes*)
    (error "Can't use cons forms in attributes: ~a" form))
  (multiple-value-bind (tag attributes body) (parse-cons-form form)
    (emit-open-tag     processor tag body attributes)
    (emit-element-body processor tag body)
    (emit-close-tag    processor tag body)))
```

In `emit-open-tag` you have to call `freshline` when appropriate and then emit the attributes with `emit-attributes`. You need to pass the element's body to `emit-open-tag` so when it's emitting XHTML, it knows whether to finish the tag with `/>` or `>`.

```
(defun emit-open-tag (processor tag body-p attributes)
  (when (or (paragraph-element-p tag) (block-element-p tag))
    (freshline processor))
  (raw-string processor (format nil "<~(~a~)" tag))
  (emit-attributes processor attributes)
  (raw-string processor (if (and *xhtml* (not body-p)) "/>" ">")))
```

In `emit-attributes` the attribute names aren't evaluated since they must be keyword symbols, but you should invoke the top-level `process` function to evaluate the attribute values, binding `*escapes*` to `*attribute-escapes*`. As a convenience for specifying boolean attributes, whose value should be the name of the attribute, if the value is `T`--not just any true value but actually `T`--then you replace the value with the name of the attribute.[7]

```
(defun emit-attributes (processor attributes)
  (loop for (k v) on attributes by #'cddr do
       (raw-string processor (format nil " ~(~a~)='" k))
       (let ((*escapes* *attribute-escapes*))
         (process processor (if (eql v t) (string-downcase k) v)))
       (raw-string processor "'")))
```

Emitting the element's body is similar to emitting the attribute values: you can loop through the body calling `process` to evaluate each form.

The rest of the code is dedicated to emitting fresh lines and adjusting the indentation as appropriate for the type of element.

```
(defun emit-element-body (processor tag body)
  (when (block-element-p tag)
    (freshline processor)
    (indent processor))
  (when (preserve-whitespace-p tag) (toggle-indenting processor))
  (dolist (item body)  (process processor item))
  (when (preserve-whitespace-p tag) (toggle-indenting processor))
  (when (block-element-p tag)
    (unindent processor)
    (freshline processor)))
```

Finally, `emit-close-tag`, as you'd probably expect, emits the closing tag (unless no closing tag is necessary, such as when the body is empty and you're either emitting XHTML or the element is one of the special empty elements). Regardless of whether you actually emit a close tag, you need to emit a final fresh line for block and paragraph elements.

```
(defun emit-close-tag (processor tag body-p)
  (unless (and (or *xhtml* (empty-element-p tag)) (not body-p))
    (raw-string processor (format nil "</~(~a~)>" tag)))
  (when (or (paragraph-element-p tag) (block-element-p tag))
    (freshline processor)))
```

The function `process` is the basic FOO interpreter. To make it a bit easier to use, you can define a function, `emit-html`, that invokes `process`, passing it an `html-pretty-printer` and a form to evaluate. You can define and use a helper function, `get-pretty-printer`, to get the pretty printer, which returns the current value of `*html-pretty-printer*` if it's bound; otherwise, it makes a new instance of `html-pretty-printer` with `*html-output*` as its output stream.

```
(defun emit-html (sexp) (process (get-pretty-printer) sexp))

(defun get-pretty-printer ()
  (or *html-pretty-printer*
      (make-instance
```

```
        'html-pretty-printer
        :printer (make-instance 'indenting-printer :out *html-output*))))
```

With this function, you can emit HTML to `*html-output*`. Rather than expose the variable `*html-output*` as part of FOO's public API, you should define a macro, `with-html-output`, that takes care of binding the stream for you. It also lets you specify whether you want pretty HTML output, defaulting to the value of the variable `*pretty*`.

```
(defmacro with-html-output ((stream &key (pretty *pretty*)) &body body)
  `(let* ((*html-output* ,stream)
          (*pretty* ,pretty))
    ,@body))
```

So, if you wanted to use `emit-html` to generate HTML to a file, you could write the following:

```
(with-open-file (out "foo.html" :direction output)
  (with-html-output (out :pretty t)
    (emit-html *some-foo-expression*)))
```

What's Next?

In the next chapter, you'll look at how to implement a macro that compiles FOO expressions into Common Lisp so you can embed HTML generation code directly into your Lisp programs. You'll also extend the FOO language to make it a bit more expressive by adding its own flavor of special operators and macros.

---

[1]In fact, it's probably *too* expressive since it can also generate all sorts of output that's not even vaguely legal HTML. Of course, that might be a feature if you need to generate HTML that's not strictly correct to compensate for buggy Web browsers. Also, it's common for language processors to accept programs that are syntactically correct and otherwise well formed that'll nonetheless provoke undefined behavior when run.

[2]Well, almost every tag. Certain tags such as `IMG` and `BR` don't. You'll deal with those in the section "The Basic Evaluation Rule."

[3]In the strict language of the Common Lisp standard, keyword symbols aren't *self-evaluating*, though they do, in fact, evaluate to themselves. See section 3.1.2.1.3 of the language standard or HyperSpec for a brief discussion.

[4]The requirement to use objects that the Lisp reader knows how to read isn't a hard-and-fast one. Since the Lisp reader is itself customizable, you could also define a new reader-level syntax for a new kind of object. But that tends to be more trouble than it's worth.

[5]Another, more purely object-oriented, approach would be to define two classes, perhaps `html-pretty-printer` and `html-raw-printer`, and then define no-op methods specialized on `html-raw-printer` for the methods that should do stuff only when `*pretty*` is true. However, in this case, after defining all the no-op methods, you'd end up with more code, and then you'd have the hassle of making sure you created an instance of the right class at the right time. But in general, using polymorphism to replace conditionals is a good strategy.

[6]You don't need a predicate for `*inline-elements*` since you only ever test for block and paragraph elements. I include the parameter here for completeness.

[7]While XHTML requires boolean attributes to be notated with their name as the value to indicate a true value, in HTML it's also legal to simply include the name of the attribute with no value, for example, **`<option selected>`** rather than **`<option selected='selected'>`**. All HTML 4.0-compatible browsers should understand both forms, but some buggy browsers understand only the no-value form for certain attributes. If you need to generate HTML for such browsers, you'll need to hack `emit-attributes` to emit those attributes a bit differently.

Copyright © 2003-2005, Peter Seibel

31. Practical: An HTML Generation Library, the Compiler

Now you're ready to look at how the FOO compiler works. The main difference between a compiler and an interpreter is that an interpreter processes a program and directly generates some behavior--generating HTML in the case of a FOO interpreter--but a compiler processes the same program and generates code in some other language that will exhibit the same behavior. In FOO, the compiler is a Common Lisp macro that translates FOO into Common Lisp so it can be embedded in a Common Lisp program. Compilers, in general, have the advantage over interpreters that, because compilation happens in advance, they can spend a bit of time optimizing the code they generate to make it more efficient. The FOO compiler does that, merging literal text as much as possible in order to emit the same HTML with a smaller number of writes than the interpreter uses. When the compiler is a Common Lisp macro, you also have the advantage that it's easy for the language understood by the compiler to contain embedded Common Lisp--the compiler just has to recognize it and embed it in the right place in the generated code. The FOO compiler will take advantage of this capability.

The Compiler

The basic architecture of the compiler consists of three layers. First you'll implement a class `html-compiler` that has one slot that holds an adjustable vector that's used to accumulate *ops* representing the calls made to the generic functions in the backend interface during the execution of `process`.

You'll then implement methods on the generic functions in the backend interface that will store the sequence of actions in the vector. Each op is represented by a list consisting of a keyword naming the operation and the arguments passed to the function that generated the op. The function `sexp->ops` implements the first phase of the compiler, compiling a list of FOO forms by calling `process` on each form with an instance of `html-compiler`.

This vector of ops stored by the compiler is then passed to a function that optimizes it, merging consecutive `raw-string` ops into a single op that emits the combined string in one go. The optimization function can also, optionally, strip out ops that are needed only for pretty printing, which is mostly important because it allows you to merge more `raw-string` ops.

Finally, the optimized ops vector is passed to a third function, `generate-code`, that returns a list of Common Lisp expressions that will actually output the HTML. When `*pretty*` is true, `generate-code` generates code that uses the methods specialized on `html-pretty-printer` to output pretty HTML. When `*pretty*` is **`NIL`**, it generates code that writes directly to the stream `*html-output*`.

The macro `html` actually generates a body that contains two expansions, one generated with `*pretty*` bound to **`T`** and one with `*pretty*` bound to

NIL. Which expansion is used is determined by the runtime value of `*pretty*`. Thus, every function that contains a call to `html` will contain code to generate both pretty and compact output.

The other significant difference between the compiler and the interpreter is that the compiler can embed Lisp forms in the code it generates. To take advantage of that, you need to modify the `process` function so it calls the `embed-code` and `embed-value` functions when asked to process an expression that's not a FOO form. Since all self-evaluating objects are valid FOO forms, the only forms that won't be passed to `process-sexp-html` are lists that don't match the syntax for FOO cons forms and non-keyword symbols, the only atoms that aren't self-evaluating. You can assume that any non-FOO cons is code to be run inline and all symbols are variables whose value you should embed.

```
(defun process (processor form)
  (cond
    ((sexp-html-p form) (process-sexp-html processor form))
    ((consp form)       (embed-code processor form))
    (t                  (embed-value processor form))))
```

Now let's look at the compiler code. First you should define two functions that slightly abstract the vector you'll use to save ops in the first two phases of compilation.

```
(defun make-op-buffer () (make-array 10 :adjustable t :fill-pointer 0))

(defun push-op (op ops-buffer) (vector-push-extend op ops-buffer))
```

Next you can define the `html-compiler` class and the methods specialized on it to implement the backend interface.

```
(defclass html-compiler ()
  ((ops :accessor ops :initform (make-op-buffer))))

(defmethod raw-string ((compiler html-compiler) string &optional newlines-p)
  (push-op `(:raw-string ,string ,newlines-p) (ops compiler)))

(defmethod newline ((compiler html-compiler))
  (push-op '(:newline) (ops compiler)))

(defmethod freshline ((compiler html-compiler))
```

```
  (push-op '(:freshline) (ops compiler)))

(defmethod indent ((compiler html-compiler))
  (push-op `(:indent) (ops compiler)))

(defmethod unindent ((compiler html-compiler))
  (push-op `(:unindent) (ops compiler)))

(defmethod toggle-indenting ((compiler html-compiler))
  (push-op `(:toggle-indenting) (ops compiler)))

(defmethod embed-value ((compiler html-compiler) value)
  (push-op `(:embed-value ,value ,*escapes*) (ops compiler)))

(defmethod embed-code ((compiler html-compiler) code)
  (push-op `(:embed-code ,code) (ops compiler)))
```

With those methods defined, you can implement the first phase of the compiler, `sexp->ops`.

```
(defun sexp->ops (body)
  (loop with compiler = (make-instance 'html-compiler)
     for form in body do (process compiler form)
     finally (return (ops compiler))))
```

During this phase you don't need to worry about the value of `*pretty*`: just record all the functions called by `process`. Here's what `sexp->ops` makes of a simple FOO form:

```
HTML> (sexp->ops '((:p "Foo")))
#((:FRESHLINE) (:RAW-STRING "<p" NIL) (:RAW-STRING ">" NIL)
  (:RAW-STRING "Foo" T) (:RAW-STRING "</p>" NIL) (:FRESHLINE))
```

The next phase, `optimize-static-output`, takes a vector of ops and returns a new vector containing the optimized version. The algorithm is simple--for each `:raw-string` op, it writes the string to a temporary string buffer. Thus, consecutive `:raw-string` ops will build up a single string containing the concatenation of the strings that need to be emitted. Whenever you encounter an op other than a `:raw-string` op, you convert the built-up string into a sequence of alternating `:raw-string` and `:newline` ops with the helper function `compile-buffer` and then add the next op. This function is also where you strip out the pretty printing ops if `*pretty*` is **`NIL`**.

```
(defun optimize-static-output (ops)
  (let ((new-ops (make-op-buffer)))
    (with-output-to-string (buf)
      (flet ((add-op (op)
               (compile-buffer buf new-ops)
               (push-op op new-ops)))
        (loop for op across ops do
             (ecase (first op)
               (:raw-string (write-sequence (second op) buf))
               ((:newline :embed-value :embed-code) (add-op op))
               ((:indent :unindent :freshline :toggle-indenting)
                (when *pretty* (add-op op)))))
        (compile-buffer buf new-ops)))
    new-ops))

(defun compile-buffer (buf ops)
  (loop with str = (get-output-stream-string buf)
     for start = 0 then (1+ pos)
     for pos = (position #\Newline str :start start)
     when (< start (length str))
     do (push-op `(:raw-string ,(subseq str start pos) nil) ops)
     when pos do (push-op '(:newline) ops)
     while pos))
```

The last step is to translate the ops into the corresponding Common Lisp code. This phase also pays attention to the value of `*pretty*`. When `*pretty*` is true, it generates code that invokes the backend generic functions on `*html-pretty-printer*`, which will be bound to an instance of `html-pretty-printer`. When `*pretty*` is **`NIL`**, it generates code that writes directly to `*html-output*`, the stream to which the pretty printer would send its output.

The actual function, `generate-code`, is trivial.

```
(defun generate-code (ops)
  (loop for op across ops collect (apply #'op->code op)))
```

All the work is done by methods on the generic function `op->code` specializing the `op` argument with an **`EQL`** specializer on the name of the op.

```
(defgeneric op->code (op &rest operands))

(defmethod op->code ((op (eql :raw-string)) &rest operands)
  (destructuring-bind (string check-for-newlines) operands
    (if *pretty*
      `(raw-string *html-pretty-printer* ,string ,check-for-newlines)
      `(write-sequence ,string *html-output*))))

(defmethod op->code ((op (eql :newline)) &rest operands)
```

```
  (if *pretty*
    `(newline *html-pretty-printer*)
    `(write-char #\Newline *html-output*)))

(defmethod op->code ((op (eql :freshline)) &rest operands)
  (if *pretty*
    `(freshline *html-pretty-printer*)
    (error "Bad op when not pretty-printing: ~a" op)))

(defmethod op->code ((op (eql :indent)) &rest operands)
  (if *pretty*
    `(indent *html-pretty-printer*)
    (error "Bad op when not pretty-printing: ~a" op)))

(defmethod op->code ((op (eql :unindent)) &rest operands)
  (if *pretty*
    `(unindent *html-pretty-printer*)
    (error "Bad op when not pretty-printing: ~a" op)))

(defmethod op->code ((op (eql :toggle-indenting)) &rest operands)
  (if *pretty*
    `(toggle-indenting *html-pretty-printer*)
    (error "Bad op when not pretty-printing: ~a" op)))
```

The two most interesting `op->code` methods are the ones that generate code for the `:embed-value` and `:embed-code` ops. In the `:embed-value` method, you can generate slightly different code depending on the value of the `escapes` operand since if `escapes` is **NIL**, you don't need to generate a call to `escape`. And when both `*pretty*` and `escapes` are **NIL**, you can generate code that uses **PRINC** to emit the value directly to the stream.

```
(defmethod op->code ((op (eql :embed-value)) &rest operands)
  (destructuring-bind (value escapes) operands
    (if *pretty*
      (if escapes
        `(raw-string *html-pretty-printer* (escape (princ-to-string ,value) ,escapes) t)
        `(raw-string *html-pretty-printer* (princ-to-string ,value) t))
      (if escapes
        `(write-sequence (escape (princ-to-string ,value) ,escapes) *html-output*)
        `(princ ,value *html-output*)))))
```

Thus, something like this:

```
HTML> (let ((x 10)) (html (:p x)))
<p>10</p>
NIL
```

works because `html` translates `(:p x)` into something like this:

```
(progn
  (write-sequence "<p>" *html-output*)
  (write-sequence (escape (princ-to-string x) "<>&") *html-output*)
  (write-sequence "</p>" *html-output*))
```

When that code replaces the call to `html` in the context of the **`LET`**, you get the following:

```
(let ((x 10))
  (progn
    (write-sequence "<p>" *html-output*)
    (write-sequence (escape (princ-to-string x) "<>&") *html-output*)
    (write-sequence "</p>" *html-output*)))
```

and the reference to `x` in the generated code turns into a reference to the lexical variable from the **`LET`** surrounding the `html` form.

The `:embed-code` method, on the other hand, is interesting because it's so trivial. Because `process` passed the form to `embed-code`, which stashed it in the `:embed-code` op, all you have to do is pull it out and return it.

```
(defmethod op->code ((op (eql :embed-code)) &rest operands)
  (first operands))
```

This allows code like this to work:

```
HTML> (html (:ul (dolist (x '(foo bar baz)) (html (:li x)))))
<ul>
  <li>FOO</li>
  <li>BAR</li>
  <li>BAZ</li>
</ul>
NIL
```

The outer call to `html` expands into code that does something like this:

```
(progn
  (write-sequence "<ul>" *html-output*)
  (dolist (x '(foo bar baz)) (html (:li x)))
  (write-sequence "</ul>" *html-output*)))
```

Then if you expand the call to `html` in the body of the **`DOLIST`**, you'll get something like this:

```
(progn
  (write-sequence "<ul>" *html-output*)
  (dolist (x '(foo bar baz))
    (progn
      (write-sequence "<li>" *html-output*)
      (write-sequence (escape (princ-to-string x) "<>&") *html-output*)
      (write-sequence "</li>" *html-output*)))
  (write-sequence "</ul>" *html-output*))
```

This code will, in fact, generate the output you saw.

FOO Special Operators

You could stop there; certainly the FOO language is expressive enough to generate nearly any HTML you'd care to. However, you can add two features to the language, with just a bit more code, that will make it quite a bit more powerful: special operators and macros.

Special operators in FOO are analogous to special operators in Common Lisp. Special operators provide ways to express things in the language that can't be expressed in the language supported by the basic evaluation rule. Or, another way to look at it is that special operators provide access to the primitive mechanisms used by the language evaluator.[1]

To take a simple example, in the FOO compiler, the language evaluator uses the `embed-value` function to generate code that will embed the value of a variable in the output HTML. However, because only symbols are passed to `embed-value`, there's no way, in the language I've described so far, to embed the value of an arbitrary Common Lisp expression; the `process` function passes cons cells to `embed-code` rather than `embed-value`, so the values returned are ignored. Typically this is what you'd want, since the main reason to embed Lisp code in a FOO program is to use Lisp control constructs. However, sometimes you'd like to embed computed values in the generated HTML. For example, you might like this FOO program to generate a paragraph tag containing a random number:

```
(:p (random 10))
```

But that doesn't work because the code is run and its value discarded.

```
HTML> (html (:p (random 10)))
<p></p>
NIL
```

In the language, as you've implemented it so far, you could work around this limitation by computing the value outside the call to `html` and then

embedding it via a variable.

```
HTML> (let ((x (random 10))) (html (:p x)))
<p>1</p>
NIL
```

But that's sort of annoying, particularly when you consider that if you could arrange for the form `(random 10)` to be passed to `embed-value` instead of `embed-code`, it'd do exactly what you want. So, you can define a special operator, `:print`, that's processed by the FOO language processor according to a different rule than a normal FOO expression. Namely, instead of generating a `<print>` element, it passes the form in its body to `embed-value`. Thus, you can generate a paragraph containing a random number like this:

```
HTML> (html (:p (:print (random 10))))
<p>9</p>
NIL
```

Obviously, this special operator is useful only in compiled FOO code since `embed-value` doesn't work in the interpreter. Another special operator that can be used in both interpreted and compiled FOO code is `:format`, which lets you generate output using the **FORMAT** function. The arguments to the `:format` special operator are a string used as a format control string and then any arguments to be interpolated. When all the arguments to `:format` are self-evaluating objects, a string is generated by passing them to **FORMAT**, and that string is then emitted like any other string. This allows such `:format` forms to be used in FOO passed to `emit-html`. In compiled FOO, the arguments to `:format` can be any Lisp expressions.

Other special operators provide control over what characters are automatically escaped and to explicitly emit newline characters: the `:noescape` special operator causes all the forms in its body to be evaluated as regular FOO forms but with `*escapes*` bound to **NIL**, while `:attribute` evaluates the forms in its body with `*escapes*` bound to

`*attribute-escapes*`. And `:newline` is translated into code to emit an explicit newline.

So, how do you define special operators? There are two aspects to processing special operators: how does the language processor recognize forms that use special operators, and how does it know what code to run to process each special operator?

You could hack `process-sexp-html` to recognize each special operator and handle it in the appropriate manner--special operators are, logically, part of the implementation of the language, and there aren't going to be that many of them. However, it'd be nice to have a slightly more modular way to add new special operators--not because users of FOO will be able to but just for your own sanity.

Define a *special form* as any list whose **CAR** is a symbol that's the name of a special operator. You can mark the names of special operators by adding a non-**NIL** value to the symbol's property list under the key `html-special-operator`. So, you can define a function that tests whether a given form is a special form like this:

```
(defun special-form-p (form)
  (and (consp form) (symbolp (car form)) (get (car form) 'html-special-operator)))
```

The code that implements each special operator is responsible for taking apart the rest of the list however it sees fit and doing whatever the semantics of the special operator require. Assuming you'll also define a function `process-special-form`, which will take the language processor and a special form and run the appropriate code to generate a sequence of calls on the processor object, you can augment the top-level `process` function to handle special forms like this:

```
(defun process (processor form)
  (cond
    ((special-form-p form) (process-special-form processor form))
    ((sexp-html-p form)    (process-sexp-html processor form))
```

```
    ((consp form)          (embed-code processor form))
    (t                     (embed-value processor form))))
```

You must add the `special-form-p` clause first because special forms can look, syntactically, like regular FOO expressions just the way Common Lisp's special forms can look like regular function calls.

Now you just need to implement `process-special-form`. Rather than define a single monolithic function that implements all the special operators, you should define a macro that allows you to define special operators much like regular functions and that also takes care of adding the `html-special-operator` entry to the property list of the special operator's name. In fact, the value you store in the property list can be a function that implements the special operator. Here's the macro:

```
(defmacro define-html-special-operator (name (processor &rest other-parameters) &body body)
  `(eval-when (:compile-toplevel :load-toplevel :execute)
     (setf (get ',name 'html-special-operator)
           (lambda (,processor ,@other-parameters) ,@body))))
```

This is a fairly advanced type of macro, but if you take it one line at a time, there's nothing all that tricky about it. To see how it works, take a simple use of the macro, the definition of the special operator `:noescape`, and look at the macro expansion. If you write this:

```
(define-html-special-operator :noescape (processor &rest body)
  (let ((*escapes* nil))
    (loop for exp in body do (process processor exp))))
```

it's as if you had written this:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf (get ':noescape 'html-special-operator)
        (lambda (processor &rest body)
          (let ((*escapes* nil))
            (loop for exp in body do (process processor exp))))))
```

The **EVAL-WHEN** special operator, as I discussed in Chapter 20, ensures that the effects of code in its body will be made visible during compilation when you compile with **COMPILE-FILE**. This matters if you want to use

`define-html-special-operator` in a file and then use the just-defined special operator in that same file.

Then the **`SETF`** expression sets the property `html-special-operator` on the symbol `:noescape` to an anonymous function with the same parameter list as was specified in `define-html-special-operator`. By defining `define-html-special-operator` to split the parameter list in two parts, `processor` and everything else, you ensure that all special operators accept at least one argument.

The body of the anonymous function is then the body provided to `define-html-special-operator`. The job of the anonymous function is to implement the special operator by making the appropriate calls on the backend interface to generate the correct HTML or the code that will generate it. It can also use `process` to evaluate an expression as a FOO form.

The `:noescape` special operator is particularly simple--all it does is pass the forms in its body to `process` with `*escapes*` bound to **`NIL`**. In other words, this special operator disables the normal character escaping preformed by `process-sexp-html`.

With special operators defined this way, all `process-special-form` has to do is look up the anonymous function in the property list of the special operator's name and **`APPLY`** it to the processor and rest of the form.

```
(defun process-special-form (processor form)
  (apply (get (car form) 'html-special-operator) processor (rest form)))
```

Now you're ready to define the five remaining FOO special operators. Similar to `:noescape` is `:attribute`, which evaluates the forms in its body with `*escapes*` bound to `*attribute-escapes*`. This special operator is useful if you want to write helper functions that output attribute values. If you write a function like this:

```
(defun foo-value (something)
  (html (:print (frob something))))
```

the `html` macro is going to generate code that escapes the characters in `*element-escapes*`. But if you're planning to use `foo-value` like this:

```
(html (:p :style (foo-value 42) "Foo"))
```

then you want it to generate code that uses `*attribute-escapes*`. So, instead, you can write it like this:[2]

```
(defun foo-value (something)
  (html (:attribute (:print (frob something)))))
```

The definition of `:attribute` looks like this:

```
(define-html-special-operator :attribute (processor &rest body)
  (let ((*escapes* *attribute-escapes*))
    (loop for exp in body do (process processor exp))))
```

The next two special operators, `:print` and `:format`, are used to output values. The `:print` special operator, as I discussed earlier, is used in compiled FOO programs to embed the value of an arbitrary Lisp expression. The `:format` special operator is more or less equivalent to generating a string with `(format nil ...)` and then embedding it. The primary reason to define `:format` as a special operator is for convenience. This:

```
(:format "Foo: ~d" x)
```

is nicer than this:

```
(:print (format nil "Foo: ~d" x))
```

It also has the slight advantage that if you use `:format` with arguments that are all self-evaluating, FOO can evaluate the `:format` at compile time rather than waiting until runtime. The definitions of `:print` and `:format` are as follows:

```
(define-html-special-operator :print (processor form)
  (cond
    ((self-evaluating-p form)
     (warn "Redundant :print of self-evaluating form ~s" form)
```

```
      (process-sexp-html processor form))
     (t
      (embed-value processor form))))

(define-html-special-operator :format (processor &rest args)
  (if (every #'self-evaluating-p args)
    (process-sexp-html processor (apply #'format nil args))
    (embed-value processor `(format nil ,@args))))
```

The `:newline` special operator forces an output of a literal newline, which is occasionally handy.

```
(define-html-special-operator :newline (processor)
  (newline processor))
```

Finally, the `:progn` special operator is analogous to the **PROGN** special operator in Common Lisp. It simply processes the forms in its body in sequence.

```
(define-html-special-operator :progn (processor &rest body)
  (loop for exp in body do (process processor exp)))
```

In other words, the following:

```
(html (:p (:progn "Foo " (:i "bar") " baz")))
```

will generate the same code as this:

```
(html (:p "Foo " (:i "bar") " baz"))
```

This might seem like a strange thing to need since normal FOO expressions can have any number of forms in their body. However, this special operator will come in quite handy in one situation--when writing FOO macros, which brings you to the last language feature you need to implement.

FOO Macros

FOO macros are similar in spirit to Common Lisp's macros. A FOO macro is a bit of code that accepts a FOO expression as an argument and returns a new FOO expression as the result, which is then evaluated according to the normal FOO evaluation rules. The actual implementation is quite similar to the implementation of special operators.

As with special operators, you can define a predicate function to test whether a given form is a macro form.

```
(defun macro-form-p (form)
  (cons-form-p form #'(lambda (x) (and (symbolp x) (get x 'html-macro)))))
```

You use the previously defined function `cons-form-p` because you want to allow macros to be used in either of the syntaxes of nonmacro FOO cons forms. However, you need to pass a different predicate function, one that tests whether the form name is a symbol with a non-**`NIL`** `html-macro` property. Also, as in the implementation of special operators, you'll define a macro for defining FOO macros, which is responsible for storing a function in the property list of the macro's name, under the key `html-macro`. However, defining a macro is a bit more complicated because FOO supports two flavors of macro. Some macros you'll define will behave much like normal HTML elements and may want to have easy access to a list of attributes. Other macros will simply want raw access to the elements of their body.

You can make the distinction between the two flavors of macros implicit: when you define a FOO macro, the parameter list can include an `&attributes` parameter. If it does, the macro form will be parsed like a regular cons form, and the macro function will be passed two values, a plist of attributes and a list of expressions that make up the body of the form. A macro form without an `&attributes` parameter won't be parsed for attributes, and the macro function will be invoked with a single argument, a list containing the body expressions. The former is useful for what are essentially HTML templates. For example:

```
(define-html-macro :mytag (&attributes attrs &body body)
  `((:div :class "mytag" ,@attrs) ,@body))

HTML> (html (:mytag "Foo"))
<div class='mytag'>Foo</div>
NIL
HTML> (html (:mytag :id "bar" "Foo"))
<div class='mytag' id='bar'>Foo</div>
NIL
HTML> (html ((:mytag :id "bar") "Foo"))
<div class='mytag' id='bar'>Foo</div>
NIL
```

The latter kind of macro is more useful for writing macros that manipulate the forms in their body. This type of macro can function as a kind of HTML control construct. As a trivial example, consider the following macro that implements an `:if` construct:

```
(define-html-macro :if (test then else)
  `(if ,test (html ,then) (html ,else)))
```

This macro allows you to write this:

```
(:p (:if (zerop (random 2)) "Heads" "Tails"))
```

instead of this slightly more verbose version:

```
(:p (if (zerop (random 2)) (html "Heads") (html "Tails")))
```

To determine which kind of macro you should generate, you need a function that can parse the parameter list given to `define-html-macro`. This function returns two values, the name of the `&attributes` parameter, or **`NIL`** if there was none, and a list containing all the elements of `args` after removing the `&attributes` marker and the subsequent list element.[3]

```
(defun parse-html-macro-lambda-list (args)
  (let ((attr-cons (member '&attributes args)))
    (values
     (cadr attr-cons)
     (nconc (ldiff args attr-cons) (cddr attr-cons)))))

HTML> (parse-html-macro-lambda-list '(a b c))
NIL
(A B C)
HTML> (parse-html-macro-lambda-list '(&attributes attrs a b c))
ATTRS
(A B C)
HTML> (parse-html-macro-lambda-list '(a b c &attributes attrs))
ATTRS
(A B C)
```

The element following `&attributes` in the parameter list can also be a destructuring parameter list.

```
HTML> (parse-html-macro-lambda-list '(&attributes (&key x y) a b c))
(&KEY X Y)
(A B C)
```

Now you're ready to write `define-html-macro`. Depending on whether there was an `&attributes` parameter specified, you need to generate one form or the other of HTML macro so the main macro simply determines which kind of HTML macro it's defining and then calls out to a helper function to generate the right kind of code.

```
(defmacro define-html-macro (name (&rest args) &body body)
  (multiple-value-bind (attribute-var args)
      (parse-html-macro-lambda-list args)
    (if attribute-var
      (generate-macro-with-attributes name attribute-var args body)
      (generate-macro-no-attributes name args body))))
```

The functions that actually generate the expansion look like this:

```
(defun generate-macro-with-attributes (name attribute-args args body)
  (with-gensyms (attributes form-body)
    (if (symbolp attribute-args) (setf attribute-args `(&rest ,attribute-args)))
    `(eval-when (:compile-toplevel :load-toplevel :execute)
       (setf (get ',name 'html-macro-wants-attributes) t)
       (setf (get ',name 'html-macro)
             (lambda (,attributes ,form-body)
               (destructuring-bind (,@attribute-args) ,attributes
                 (destructuring-bind (,@args) ,form-body
                   ,@body)))))))

(defun generate-macro-no-attributes (name args body)
  (with-gensyms (form-body)
    `(eval-when (:compile-toplevel :load-toplevel :execute)
       (setf (get ',name 'html-macro-wants-attributes) nil)
       (setf (get ',name 'html-macro)
             (lambda (,form-body)
               (destructuring-bind (,@args) ,form-body ,@body))))))
```

The macro functions you'll define accept either one or two arguments and then use **DESTRUCTURING-BIND** to take them apart and bind them to the parameters defined in the call to `define-html-macro`. In both expansions you need to save the macro function in the name's property list under `html-macro` and a boolean indicating whether the macro takes an `&attributes` parameter under the property `html-macro-wants-attributes`. You use that property in the following function, `expand-macro-form`, to determine how the macro function should be invoked:

```
(defun expand-macro-form (form)
  (if (or (consp (first form))
          (get (first form) 'html-macro-wants-attributes))
    (multiple-value-bind (tag attributes body) (parse-cons-form form)
      (funcall (get tag 'html-macro) attributes body))
    (destructuring-bind (tag &body body) form
      (funcall (get tag 'html-macro) body))))
```

The last step is to integrate macros by adding a clause to the dispatching **COND** in the top-level `process` function.

```
(defun process (processor form)
  (cond
    ((special-form-p form) (process-special-form processor form))
```

```
((macro-form-p form)    (process processor (expand-macro-form form)))
((sexp-html-p form)     (process-sexp-html processor form))
((consp form)           (embed-code processor form))
(t                      (embed-value processor form))))
```

This is the final version of `process`.

The Public API

Now, at long last, you're ready to implement the `html` macro, the main entry point to the FOO compiler. The other parts of FOO's public API are `emit-html` and `with-html-output`, which I discussed in the previous chapter, and `define-html-macro`, which I discussed in the previous section. The `define-html-macro` macro needs to be part of the public API because FOO's users will want to write their own HTML macros. On the other hand, `define-html-special-operator` isn't part of the public API because it requires too much knowledge of FOO's internals to define a new special operator. And there should be very little that can't be done using the existing language and special operators.[4]

One last element of the public API, before I get to `html`, is another macro, `in-html-style`. This macro controls whether FOO generates XHTML or regular HTML by setting the `*xhtml*` variable. The reason this needs to be a macro is because you'll want to wrap the code that sets `*xhtml*` in an **`EVAL-WHEN`** so you can set it in a file and have it affect uses of the `html` macro later in that same file.

```
(defmacro in-html-style (syntax)
  (eval-when (:compile-toplevel :load-toplevel :execute)
    (case syntax
      (:html (setf *xhtml* nil))
      (:xhtml (setf *xhtml* t)))))
```

Finally let's look at `html` itself. The only tricky bit about implementing `html` comes from the need to generate code that can be used to generate both pretty and compact output, depending on the runtime value of the variable `*pretty*`. Thus, `html` needs to generate an expansion that contains an **`IF`** expression and two versions of the

code, one compiled with `*pretty*` bound to true and one compiled with it bound to **NIL**. To further complicate matters, it's common for one `html` call to contain embedded calls to `html`, like this:

```
(html (:ul (dolist (item stuff)) (html (:li item))))
```

If the outer `html` expands into an **IF** expression with two versions of the code, one for when `*pretty*` is true and one for when it's false, it's silly for nested `html` forms to expand into two versions too. In fact, it'll lead to an exponential explosion of code since the nested `html` is already going to be expanded twice--once in the `*pretty*`-is-true branch and once in the `*pretty*`-is-false branch. If each expansion generates two versions, then you'll have four total versions. And if the nested `html` form contained another nested `html` form, you'd end up with eight versions of that code. If the compiler is smart, it'll eventually realize that most of that generated code is dead and will eliminate it, but even figuring that out can take quite a bit of time, slowing down compilation of any function that uses nested calls to `html`.

Luckily, you can easily avoid this explosion of dead code by generating an expansion that locally redefines the `html` macro, using **MACROLET**, to generate only the right kind of code. First you define a helper function that takes the vector of ops returned by `sexp->ops` and runs it through `optimize-static-output` and `generate-code`--the two phases that are affected by the value of `*pretty*`--with `*pretty*` bound to a specified value and that interpolates the resulting code into a **PROGN**. (The **PROGN** returns **NIL** just to keep things tidy.).

```
(defun codegen-html (ops pretty)
  (let ((*pretty* pretty))
    `(progn ,@(generate-code (optimize-static-output ops)) nil)))
```

With that function, you can then define `html` like this:

```
(defmacro html (&whole whole &body body)
  (declare (ignore body))
  `(if *pretty*
     (macrolet ((html (&body body) (codegen-html (sexp->ops body) t)))
       (let ((*html-pretty-printer* (get-pretty-printer))) ,whole))
     (macrolet ((html (&body body) (codegen-html (sexp->ops body) nil)))
       ,whole)))
```

The **`&whole`** parameter represents the original `html` form, and because it's interpolated into the expansion in the bodies of the two **`MACROLET`**s, it will be reprocessed with each of the new definitions of `html`, the one that generates pretty-printing code and the other that generates non-pretty-printing code. Note that the variable `*pretty*` is used both during macro expansion *and* when the resulting code is run. It's used at macro expansion time by `codegen-html` to cause `generate-code` to generate one kind of code or the other. And it's used at runtime, in the **`IF`** generated by the top-level `html` macro, to determine whether the pretty-printing or non-pretty-printing code should actually run.

The End of the Line

As usual, you could keep working with this code to enhance it in various ways. One interesting avenue to pursue is to use the underlying output generation framework to emit other kinds of output. In the version of FOO you can download from the book's Web site, you'll find some code that implements CSS output that can be integrated into HTML output in both the interpreter and compiler. That's an interesting case because CSS's syntax can't be mapped to s-expressions in such a trivial way as HTML's can. However, if you look at that code, you'll see it's still possible to define an s-expression syntax for representing the various constructs available in CSS.

A more ambitious undertaking would be to add support for generating embedded JavaScript. Done right, adding JavaScript support to FOO could yield two big wins. One is that after you define an s-expression syntax that you can map to JavaScript syntax, then you can start writing macros, in Common Lisp, to add new constructs to the language you use to write client-side code, which will then be compiled to JavaScript. The other is that, as part of the FOO s-expression JavaScript to regular JavaScript translation, you could deal with the subtle but annoying differences between JavaScript implementations in different browsers. That is, the JavaScript code that FOO generates could either contain the appropriate conditional code to do one thing in one browser and another in a different browser or could generate different code depending on which browser you wanted to support. Then if you use FOO in dynamically

generated pages, it could use information about the User-Agent making the request to generate the right flavor of JavaScript for that browser.

But if that interests you, you'll have to implement it yourself since this is the end of the last practical chapter of this book. In the next chapter I'll wrap things up, discussing briefly some topics that I haven't touched on elsewhere in the book such as how to find libraries, how to optimize Common Lisp code, and how to deliver Lisp applications.

---

[1]The analogy between FOO's special operators, and macros, which I'll discuss in the next section, and Lisp's own is fairly sound. In fact, understanding how FOO's special operators and macros work may give you some insight into why Common Lisp is put together the way it is.

[2]The `:noescape` and `:attribute` special operators must be defined as special operators because FOO determines what escapes to use at compile time, not at runtime. This allows FOO to escape literal values at compile time, which is much more efficient than having to scan all output at runtime.

[3]Note that `&attributes` is just another symbol; there's nothing intrinsically special about names that start with `&`.

[4]The one element of the underlying language-processing infrastructure that's not currently exposed through special operators is the indentation. If you wanted to make FOO more flexible, albeit at the cost of making its API that much more complex, you could add special operators for manipulating the underlying indenting printer. But it seems like the cost of having to explain the extra special operators would outweigh the rather small gain in expressiveness.

Copyright © 2003-2005, Peter Seibel

32. Conclusion: What's Next?

I hope by now you're convinced that the title of this book isn't an oxymoron. However, it's quite likely there's some area of programming that's of great practical importance to you that I haven't discussed at all. For instance, I haven't said anything about how to develop graphical user interfaces (GUIs), how to connect to relational databases, how to parse XML, or how to write programs that act as clients for various network protocols. Similarly, I haven't discussed two topics that will become important when you write real applications in Common Lisp: optimizing your Lisp code and packaging your application for delivery.

I'm obviously not going to cover all these topics in depth in this final chapter. Instead, I'll give you a few pointers you can use to pursue whichever aspect of Lisp programming interests you most.

Finding Lisp Libraries

While the standard library of functions, data types, and macros that comes with Common Lisp is quite large, it provides only general-purpose programming constructs. Specialized tasks such as writing GUIs, talking to databases, and parsing XML require libraries beyond what are provided by the ANSI standardized language.

The easiest way to obtain a library to do something you need may be simply to check out your Lisp implementation. Most implementations provide at least some facilities not specified in the language standard. The commercial Common Lisp vendors tend to work especially hard at providing additional libraries for their implementation in order to justify their prices. Franz's Allegro Common Lisp, Enterprise Edition, for instance, comes with libraries for parsing XML, speaking SOAP, generating HTML, connecting to relational databases, and building graphical interfaces in various ways, among others. LispWorks, another prominent commercial Lisp, provides several similar libraries, including a well-regarded portable GUI toolkit, CAPI, which can be used to develop GUI applications that will run on any operating system LispWorks runs on.

The free and open-source Common Lisp implementations typically don't include quite so many bundled libraries, relying instead on portable free and open-source libraries. But even those implementations usually fill in some of the more important areas not addressed by the language standard such as networking and multithreading.

The only disadvantage of using implementation-specific libraries is that they tie you to the implementation that provides them. If you're delivering end-user apps or are deploying a server-based application on a server that you control, that may not matter a lot. But if you want to write code to share with other Lispers or if you simply don't want to be tied to a particular implementation, it's a little more annoying.

For portable libraries--portable either because they're written entirely in standard Common Lisp or because they contain appropriate read-time conditionalization to work on multiple implementations[1]--your best bet is to go to the Web. With the usual caveats about URLs going stale as soon as they're printed on paper, these are three of the best current starting points:

- Common-Lisp.net (`http://www.common-lisp.net/`) is a site that hosts free and open-source Common Lisp projects, providing version control, mailing lists, and Web hosting of project pages. In the first year and a half after the site went live, nearly a hundred projects were registered.
- The Common Lisp Open Code Collection (CLOCC) (`http://clocc.sourceforge.net/`) is a slightly older collection of free software libraries, which are intended to be portable between Common Lisp implementations and self-contained, not relying on any libraries not included in CLOCC itself.
- Cliki (`http://www.cliki.net/`) is a wiki devoted to free software in Common Lisp. While, like any wiki, it may change at any time, typically it has quite a few links to libraries as well to various open-source Common Lisp implementations. The eponymous software it runs on is also written in Common Lisp.

Linux users running the Debian or Gentoo distributions can also easily install an ever-growing number of Lisp libraries that have been packaged with those distributions' packing tools, `apt-get` on Debian and `emerge` on Gentoo.

I won't recommend any specific libraries here since the library situation is changing every day--after years of envying the library collections of Perl, Python, and Java, Common Lispers have, in the past couple of years, begun to take up the challenge of giving Common Lisp the set of libraries--both open source and commercial--that it deserves.

One area where there has been a lot of activity recently is on the GUI front. Unlike Java and C# but like Perl, Python, and C, there's no single way to develop GUIs in Common Lisp. Instead, it depends both on what Common Lisp implementation you're using and what operating system or systems you want to support.

The commercial Common Lisp implementations usually provide some way to build GUIs for the platforms they run on. Additionally, LispWorks provides CAPI, the previously mentioned, portable GUI API.

On the open-source side, you have a number of options. On Unix, you can write low-level X Windows GUIs using CLX, a pure-Common Lisp implementation of the X Windows protocol, roughly akin to xlib in C. Or you can use various bindings to higher-level APIs and toolkits such as GTK and Tk, much the way you might in Perl or Python.

Or, if you're looking for something completely different, you can check out Common Lisp Interface Manager (CLIM). A descendant of the Symbolics Lisp Machines GUI framework, CLIM is powerful but complex. Although many commercial Common Lisp implementations actually support it, it doesn't seem to have seen a lot of use. But in the past couple years, an open-source implementation of CLIM, McCLIM--now hosted at Common-Lisp.net--has been picking up steam lately, so we may be on the verge of a CLIM renaissance.

Interfacing with Other Languages

While many useful libraries can be written in "pure" Common Lisp using only the features specified in the language standard, and many more can be written in Lisp using nonstandard facilities provided by a given implementation, occasionally it's more straightforward to use an existing library written in another language, such as C.

The language standard doesn't specify a mechanism for Lisp code to call code written in another language or even require that implementations provide such a mechanism. But these days, almost all Common Lisp implementations support what's called a *Foreign Function Interface*, or FFI for short.[2] The basic job of an FFI is to allow you to give Lisp enough information to be able to link in the foreign code. Thus, if you're going to call a function from a C library, you need to tell Lisp about how to translate the Lisp objects passed to the function into C types and the value returned by the function back into a Lisp object. However, each implementation provides its own FFI, each with slightly varying capabilities and syntax. Some FFIs allow callbacks from C to Lisp, and others don't. The Universal Foreign Function Interface (UFFI) project provides a portability layer over the FFIs of more than a half dozen different Common Lisp implementations. It works by defining its own macros that expand into appropriate FFI code for the implementation it's running in. The UFFI takes a lowest common denominator approach, which means it can't take advantage of all the features of different implementations' FFIs, but it does provide a good way to build a simple Lisp wrapper around a basic C API.[3]

Make It Work, Make It Right, Make It Fast

As has been said many times, and variously attributed to Donald Knuth, C.A.R. Hoare, and Edsger Dijkstra, premature optimization is the root of all evil.[4] Common Lisp is an excellent language to program in if you want to heed this wisdom yet still need high performance. This may come as a surprise if you've heard the conventional wisdom that Lisp is slow. In Lisp's earliest days, when computers were programmed with punch cards, Lisp's high-level features may have doomed it to be slower than the competition, namely, assembly and FORTRAN. But that was a long time ago. In the meantime, Lisp has been used for everything from creating complex AI systems to writing operating systems, and a lot of work has gone into figuring out how to compile Lisp into efficient code. In this section I'll talk about some of the reasons why Common Lisp is an excellent language for writing high-performance code and some of the techniques for doing so.

The first reason that Lisp is an excellent language for writing high-performance code is, ironically enough, the dynamic nature of Lisp programming--the very thing that originally made it hard to bring Lisp's performance up to the levels achieved by FORTRAN compilers. The reason Common Lisp's dynamic features make it easier to write high-performance code is that the first step to writing efficient code is to find the right algorithms and data structures.

Common Lisp's dynamic features keep code flexible, which makes it easier to try different approaches. Given a finite amount of time to write a program, you're much more likely to end up with a high-performance version if you don't spend a lot of time getting into and out of dead ends. In Common Lisp, you can try an idea, see it's going nowhere, and move on without having spent a ton of time convincing the compiler your code is worthy of being run and then waiting for it to finish compiling. You can write a straightforward but inefficient version of a function--a *code sketch*--to determine whether your basic approach is sound and then replace that function with a more complex but more efficient implementation if you determine that it is. And if the overall approach turns out

to be flawed, then you haven't wasted a bunch of time tuning a function that's no longer needed, which means you have more time to find a better approach.

The next reason Common Lisp is a good language for developing high-performance software is that most Common Lisp implementations come with mature compilers that generate quite efficient machine code. I'll talk in a moment about how to help these compilers generate code that will be competitive with code generated by C compilers, but these implementations already are quite a bit faster than those of languages whose implementations are less mature and use simpler compilers or interpreters. Also, since the Lisp compiler is available at runtime, the Lisp programmer has some possibilities that would be hard to emulate in other languages--your programs can generate Lisp code at runtime that's then compiled into machine code and run. If the generated code is going to run enough times, this can be a big win. Or, even without using the compiler at runtime, closures give you another way to meld machine code with runtime data. For instance, the CL-PPCRE regular expression library, running in CMUCL, is faster than Perl's regular expression engine on some benchmarks, even though Perl's engine is written in highly tuned C. This is presumably because in Perl a regular expression is translated into what are essentially bytecodes that are then interpreted by the regex engine, while CL-PPCRE translates a regular expression into a tree of compiled closures that invoke each other via the normal function-calling machinery.[5]

However, even with the right algorithm and a high-quality compiler, you may not get the raw speed you need. Then it's time to think about profiling and tuning. The key, in Lisp as in any language, is to profile first to find the spots where your program is actually spending its time and then worry about speeding up those parts.[6]

You have a number of different ways to approach profiling. The language standard provides a few rudimentary tools for measuring how long certain forms take to execute. In particular, the `TIME` macro can be wrapped around any form and will return whatever values the form returns after printing a message to

TRACE-OUTPUT about how long it took to run and how much memory it used. The exact form of the message is implementation defined.

You can use **TIME** for a bit of quick-and-dirty profiling to narrow your search for bottlenecks. For instance, suppose you have a function that's taking a long time to run and that calls two other functions--something like this:

```
(defun foo ()
  (bar)
  (baz))
```

If you want to see whether `bar` or `baz` is taking more time, you can change the definition of `foo` to this:

```
(defun foo ()
  (time (bar))
  (time (baz)))
```

Now you can call `foo`, and Lisp will print two reports, one for `bar` and one for `baz`. The form is implementation dependent; here's what it looks like in Allegro Common Lisp:

```
CL-USER> (foo)
; cpu time (non-gc) 60 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  60 msec user, 0 msec system
; real time  105 msec
; space allocation:
;  24,172 cons cells, 1,696 other bytes, 0 static bytes
; cpu time (non-gc) 540 msec user, 10 msec system
; cpu time (gc)     170 msec user, 0 msec system
; cpu time (total)  710 msec user, 10 msec system
; real time  1,046 msec
; space allocation:
;  270,172 cons cells, 1,696 other bytes, 0 static bytes
```

Of course, that'd be a bit easier to read if the output included a label. If you use this technique a lot, it might be worth defining your own macro like this:

```
(defmacro labeled-time (form)
  `(progn
    (format *trace-output* "~2&~a" ',form)
    (time ,form)))
```

If you replace **TIME** with `labeled-time` in `foo`, you'll get this output:

```
CL-USER> (foo)

(BAR)
; cpu time (non-gc) 60 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  60 msec user, 0 msec system
```

```
; real time  131 msec
; space allocation:
;  24,172 cons cells, 1,696 other bytes, 0 static bytes

(BAZ)
; cpu time (non-gc) 490 msec user, 0 msec system
; cpu time (gc)     190 msec user, 10 msec system
; cpu time (total)  680 msec user, 10 msec system
; real time  1,088 msec
; space allocation:
;  270,172 cons cells, 1,696 other bytes, 0 static bytes
```

From this output, it's clear that most of the time in `foo` is spent in `baz`.

Of course, the output from **`TIME`** gets a bit unwieldy if the form you want to profile is called repeatedly. You can build your own measurement tools using the functions **`GET-INTERNAL-REAL-TIME`** and **`GET-INTERNAL-RUN-TIME`**, which return a number that increases by the value of the constant **`INTERNAL-TIME-UNITS-PER-SECOND`** each second. **`GET-INTERNAL-REAL-TIME`** measures *wall time*, the actual amount of time elapsed, while **`GET-INTERNAL-RUN-TIME`** measures some implementation-defined value such as the amount of time Lisp was actually executing or the time Lisp was executing user code and not internal bookkeeping such as the garbage collector. Here's a trivial but useful profiling tool built with a few macros and **`GET-INTERNAL-RUN-TIME`**:

```
(defparameter *timing-data* ())

(defmacro with-timing (label &body body)
  (with-gensyms (start)
    `(let ((,start (get-internal-run-time)))
      (unwind-protect (progn ,@body)
        (push (list ',label ,start (get-internal-run-time)) *timing-data*)))))

(defun clear-timing-data ()
  (setf *timing-data* ()))

(defun show-timing-data ()
  (loop for (label time count time-per %-of-total) in (compile-timing-data) do
       (format t "~3d% ~a: ~d ticks over ~d calls for ~d per.~%"
               %-of-total label time count time-per)))

(defun compile-timing-data ()
  (loop with timing-table = (make-hash-table)
     with count-table = (make-hash-table)
     for (label start end) in *timing-data*
     for time = (- end start)
     summing time into total
     do
       (incf (gethash label timing-table 0) time)
       (incf (gethash label count-table 0))
     finally
       (return
         (sort
          (loop for label being the hash-keys in timing-table collect
               (let  ((time (gethash label timing-table))
```

```
                (count (gethash label count-table)))
           (list label time count (round (/ time count)) (round (* 100 (/ time total))))))
      #'> :key #'fifth))))
```

This profiler lets you wrap a `with-timing` around any form; each time the form is executed, the time it starts and the time it ends are recorded, associating with a label you provide. The function `show-timing-data` dumps out a table showing how much time was spent in different labeled sections of code like this:

```
CL-USER> (show-timing-data)
 84% BAR: 650 ticks over 2 calls for 325 per.
 16% FOO: 120 ticks over 5 calls for 24 per.
NIL
```

You could obviously make this profiling code more sophisticated in many ways. Alternatively, your Lisp implementation most likely provides its own profiling tools, which, since they have access to the internals of the implementation, can get at information not necessarily available to user-level code.

Once you've found the bottleneck in your code, you can start tuning. The first thing you should try, of course, is to find a more efficient basic algorithm--that's where the big gains are to be had. But assuming you're already using an appropriate algorithm, then it's down to *code bumming*--locally optimizing the code so it does absolutely no more work than necessary.

The main tools for code bumming in Common Lisp are its optional declarations. The basic idea behind declarations in Common Lisp is that they're used to give the compiler information it can use in a variety of ways to generate better code.

For a simple example, consider this Common Lisp function:

```
(defun add (x y) (+ x y))
```

I mentioned in Chapter 10 that if you compare the performance of this function Lisp to the seemingly equivalent C function:

```
int add (int x, int y) { return x + y; }
```

you'll likely find the Common Lisp version to be quite a bit slower, even if your Common Lisp implementation features a high-quality native compiler.

That's because the Common Lisp version is doing a lot more--the Common Lisp compiler doesn't even know that the values of `a` and `b` are numbers and so has to generate code to check at runtime. And once it determines they *are* numbers, it has to determine what types of numbers--integers, rationals, floating point, or complex--and dispatch to the appropriate addition routine for the actual types. And even if `a` and `b` are integers--the case you care about--then the addition routine has to account for the possibility that the result may be too large to represent as a *fixnum*, a number that can be represented in a single machine word, and thus it may have to allocate a *bignum* object.

In C, on the other hand, because the type of all variables are declared, the compiler knows exactly what kind of values `a` and `b` will hold. And because C's arithmetic simply overflows when the result of an addition is too large to represent in whatever type is being returned, there's no checking for overflow and no allocation of a bignum object to represent the result when the mathematical sum is too large to fit in a machine word.

Thus, while the behavior of the Common Lisp code is much more likely to be mathematically correct, the C version can probably be compiled down to one or two machine instructions. But if you're willing to give the Common Lisp compiler the same information the C compiler has about the types of arguments and return values and to accept certain C-like compromises in terms of generality and error checking, the Common Lisp function can also be compiled down to an instruction or two.

That's what declarations are for. The main use of declarations is to tell the compiler about the types of variables and other expressions. For instance, you could tell the compiler that the arguments to `add` are both fixnums by writing the function like this:

```
(defun add (x y)
  (declare (fixnum x y))
  (+ x y))
```

The **`DECLARE`** expression isn't a Lisp form; rather, it's part of the syntax of the **`DEFUN`** and must appear before any other code in the function body.[7] This

declaration declares that the arguments passed for the parameters `x` and `y` will always be fixnums. In other words, it's a promise to the compiler, and the compiler is allowed to generate code on the assumption that whatever you tell it is true.

To declare the type of the value returned, you can wrap the form `(+ x y)` in the **THE** special operator. This operator takes a type specifier, such as **FIXNUM**, and a form and tells the compiler the form will evaluate to the given type. Thus, to give the Common Lisp compiler all the information about `add` that the C compiler gets, you can write it like this:

```
(defun add (x y)
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

However, even this version needs one more declaration to give the Common Lisp compiler the same license as the C compiler to generate fast but dangerous code. The **OPTIMIZE** declaration is used to tell the compiler how to balance five qualities: the speed of the code generated; the amount of runtime error checking; the memory usage of the code, both in terms of code size and runtime memory usage; the amount of debugging information kept with the code; and the speed of the compilation process. An **OPTIMIZE** declaration consists of one or more lists, each containing one of the symbols **SPEED**, **SAFETY**, **SPACE**, **DEBUG**, and **COMPILATION-SPEED**, and a number from zero to three, inclusive. The number specifies the relative weighting the compiler should give to the corresponding quality, with `3` being the most important and `0` meaning not important at all. Thus, to make Common Lisp compile `add` more or less like a C compiler would, you can write it like this:

```
(defun add (x y)
  (declare (optimize (speed 3) (safety 0)))
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

Of course, now the Lisp version suffers from many of the same liabilities as the C version--if the arguments passed aren't fixnums or if the addition overflows, the result will be mathematically incorrect or worse. Also, if someone calls `add` with a wrong number of arguments, it may not be pretty. Thus, you should use

these kinds of declarations only after your program is working correctly. And you should add them only where profiling shows they'll make a difference. If you're getting reasonable performance without them, leave them out. But when profiling shows you a real hot spot in your code and you need to tune it up, go ahead. Because you can use declarations this way, it's rarely necessary to rewrite code in C just for performance reasons; FFIs are used to access existing C code, but declarations are used when C-like performance is needed. Of course, how close you can get the performance of a given piece of Common Lisp code to C and C++ depends mostly on how much like C you're willing to make it.

Another code-tuning tool built into Lisp is the function **DISASSEMBLE**. The exact behavior of this function is implementation dependent because it depends on how the implementation compiles code--whether to machine code, bytecodes, or some other form. But the basic idea is that it shows you the code generated by the compiler when it compiled a specific function.

Thus, you can use **DISASSEMBLE** to see whether your declarations are having any effect on the code generated. And if your Lisp implementation uses a native compiler and you know your platform's assembly language, you can get a pretty good sense of what's actually going on when you call one of your functions. For instance, you could use **DISASSEMBLE** to get a sense of the difference between the first version of `add`, with no declarations, and the final version. First, define and compile the original version.

```
(defun add (x y) (+ x y))
```

Then, at the REPL, call **DISASSEMBLE** with the name of the function. In Allegro, it shows the following assembly-language-like dump of the code generated by the compiler:

```
CL-USER> (disassemble 'add)
;; disassembly of #<Function ADD>
;; formals: X Y

;; code start: #x737496f4:
   0: 55         pushl  ebp
   1: 8b ec    movl ebp,esp
   3: 56         pushl  esi
   4: 83 ec 24 subl esp,$36
```

```
   7: 83 f9 02 cmpl ecx,$2
  10: 74 02    jz   14
  12: cd 61    int  $97   ; SYS::TRAP-ARGERR
  14: 80 7f cb 00 cmpb  [edi-53],$0        ; SYS::C_INTERRUPT-PENDING
  18: 74 02    jz   22
  20: cd 64    int  $100  ; SYS::TRAP-SIGNAL-HIT
  22: 8b d8    movl ebx,eax
  24: 0b da    orl  ebx,edx
  26: f6 c3 03 testb    bl,$3
  29: 75 0e    jnz  45
  31: 8b d8    movl ebx,eax
  33: 03 da    addl ebx,edx
  35: 70 08    jo   45
  37: 8b c3    movl eax,ebx
  39: f8         clc
  40: c9         leave
  41: 8b 75 fc movl esi,[ebp-4]
  44: c3         ret
  45: 8b 5f 8f movl ebx,[edi-113]    ; EXCL::+_2OP
  48: ff 57 27 call *[edi+39]   ; SYS::TRAMP-TWO
  51: eb f3    jmp  40
  53: 90         nop
; No value
```

Clearly, there's a bunch of stuff going on here. If you're familiar with x86 assembly language, you can probably tell what. Now compile this version of `add` with all the declarations.

```
(defun add (x y)
  (declare (optimize (speed 3) (safety 0)))
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

Now disassemble `add` again, and see if the declarations had any effect.

```
CL-USER> (disassemble 'add)
;; disassembly of #<Function ADD>
;; formals: X Y

;; code start: #x7374dc34:
   0: 03 c2    addl eax,edx
   2: f8         clc
   3: 8b 75 fc movl esi,[ebp-4]
   6: c3         ret
   7: 90         nop
; No value
```

Looks like they did.

Delivering Applications

Another topic of practical importance, which I didn't talk about elsewhere in the book, is how to deliver software written in Lisp. The main reason I neglected this topic is because there are many different ways to do it, and which one is best for you depends on what kind of software you need to deliver to what kind of user with what Common Lisp implementation. In this section I'll give an overview of some of the different options.

If you've written code you want to share with fellow Lisp programmers, the most straightforward way to distribute it is as source code.[8] You can distribute a simple library as a single source file, which programmers can `LOAD` into their Lisp image, possibly after compiling it with `COMPILE-FILE`.

More complex libraries or applications, broken up across multiple source files, pose an additional challenge--in order to load and compile the code, the files need to be loaded and compiled in the correct order. For instance, a file containing macro definitions must be loaded before you can compile files that use those macros. And a file containing `DEFPACKAGE` forms must be loaded before any files that use those packages can even be `READ`. Lispers call this the *system definition* problem and typically handle it with tools called *system definition facilities* or *system definition utilities*, which are somewhat analogous to build tools such as `make` or `ant`. As with `make` and `ant`, system definition tools allow you to specify the dependencies between different files and then take care of loading and compiling

the files in the correct order while trying to do only work that's necessary--recompiling only files that have changed, for example.

These days the most widely used system definition tool is ASDF, which stands for *Another System Definition Facility*.[9] The basic idea behind ASDF is that you define systems in ASD files, and ASDF provides a number of operations on systems such as loading them or compiling them. A system can also be defined to depend on other systems, which will be loaded as necessary. For instance, the following shows the contents of `html.asd`, the ASD file for the FOO library from Chapters 31 and 32:

```
(defpackage :com.gigamonkeys.html-system (:use :asdf :cl))
(in-package :com.gigamonkeys.html-system)

(defsystem html
  :name "html"
  :author "Peter Seibel <peter@gigamonkeys.com>"
  :version "0.1"
  :maintainer "Peter Seibel <peter@gigamonkeys.com>"
  :license "BSD"
  :description "HTML and CSS generation from sexps."
  :long-description ""
  :components
  ((:file "packages")
   (:file "html" :depends-on ("packages"))
   (:file "css" :depends-on ("packages" "html")))
  :depends-on (:macro-utilities))
```

If you add a symbolic link to this file from a directory listed in `asdf:*central-registry*`,[10] then you can type this:

```
(asdf:operate 'asdf:load-op :html)
```

to compile and load the files `packages.lisp`, `html.lisp`, and `html-macros.lisp` in the correct order after first making sure the `:macro-utilities` system has been compiled and loaded. For other examples of ASD files, you can look at this book's source code--the

code from each practical chapter is defined as a system with appropriate intersystem dependencies expressed in the ASD files.

Most free and open-source Common Lisp libraries you'll find will come with an ASD file. Some will use other system definition tools such as the slightly older MK:DEFSYSTEM or even utilities devised by the library's author, but the tide seems to be turning in the direction of ASDF.[11]

Of course, while ASDF makes it easy for Lispers to install Lisp libraries, it's not much help if you want to package an application for an end user who doesn't know or care about Lisp. If you're delivering a pure end-user application, presumably you want to provide something the user can download, install, and run without having to know anything about Lisp. You can't expect them to separately download and install a Lisp implementation. And you want them to be able to run your application just like any other application--by double-clicking an icon on Windows or OS X or by typing the name of the program at the command line on Unix.

However, unlike C programs, which can typically rely on certain shared libraries (DLLs on Windows) that make up the C "runtime" being present as part of the operating system, Lisp programs must include a Lisp runtime, that is, the same program you run when you start Lisp though perhaps with certain functionality not needed to run the application excised.

To further complicate matters, *program* isn't really well defined in Lisp. As you've seen throughout this book, the process of developing software in Lisp is an incremental process that involves making

changes to the set of definitions and data living in your Lisp image. The "program" is just a particular state of the image arrived at by loading the `.lisp` or `.fasl` files that contain code that creates the appropriate definitions and data. You could, then, distribute a Lisp application as a Lisp runtime plus a bunch of FASL files and an executable that starts the runtime, loads the FASLs, and somehow invokes the appropriate starting function. However, since actually loading the FASLs can take some time, especially if they have to do any computation to set up the state of the world, most Common Lisp implementations provide a way to *dump an image*--to save the state of a running Lisp to a file called an *image file* or sometimes a *core*. When a Lisp runtime starts, the first thing it does is load an image file, which it can do in much less time than it'd take to re-create the state by loading FASL files.

Normally the image file is a default image containing only the standard packages defined by the language and any extras provided by the implementation. But with most implementations, you have a way to specify a different image file. Thus, instead of packaging an app as a Lisp runtime plus a bunch of FASLs, you can package it as a Lisp runtime plus a single image file containing all the definitions and data that make up your application. Then all you need is a program that launches the Lisp runtime with the appropriate image file and invokes whatever function serves as the entry point to the application.

This is where things get implementation and operating-system dependent. Some Common Lisp implementations, in particular the commercial ones such as Allegro and LispWorks, provide tools for

building such an executable. For instance, Allegro's Enterprise Edition provides a function `excl:generate-application` that creates a directory containing the Lisp runtime as a shared library, an image file, and an executable that starts the runtime with the given image. Similarly, the LispWorks Professional Edition "delivery" mechanism allows you to build single-file executables of your programs. On Unix, with the various free and open-source implementations, you can do essentially the same thing except it's probably easier to use a shell script to start everything.

And on OS X things are even better--since all applications on OS X are packaged as `.app` bundles, which are essentially directories with a certain structure, it's not all that difficult to package all the parts of a Lisp application as a double-clickable `.app` bundle. Mikel Evins's Bosco tool makes it easy to create `.app` bundles for applications running on OpenMCL.

Of course, another popular way to deliver applications these days is as server-side applications. This is a niche where Common Lisp can really excel--you can pick a combination of operating system and Common Lisp implementation that works well for you, and you don't have to worry about packaging the application to be installed by an end user. And Common Lisp's interactive debugging and development features make it possible to debug and upgrade a live server in ways that either just aren't possible in a less dynamic language or would require you to build a lot of specific infrastructure.

Where to Go Next

So, that's it. Welcome to the wonderful world of Lisp. The best thing you can do now--if you haven't already--is to start writing your own Lisp code. Pick a project that interests you, and do it in Common Lisp. Then do another. Lather, rinse, repeat.

However, if you need some further pointers, this section offers some places to go. For starters, check out the *Practical Common Lisp* Web site at `http://www.gigamonkeys.com/book/`, where you can find the source code from the practical chapters, errata, and links to other Lisp resources on the Web.

In addition to the sites I mentioned in the "Finding Lisp Libraries" section, you may also want explore the Common Lisp HyperSpec (a.k.a. the HyperSpec or CLHS), an HTML version of the ANSI language standard prepared by Kent Pitman and made available by LispWorks at `http://www.lispworks.com/documentation/HyperSpec/index.html`. The HyperSpec is by no means a tutorial, but it's as authoritative a guide to the language as you can get without buying a printed copy of the standard from ANSI and much more convenient for day-to-day use.[12]

If you want to get in touch with other Lispers, `comp.lang.lisp` on Usenet and the `#lisp` IRC channel or the Freenode network (`http://www.freenode.net`) are two of the main online hang- outs. There are also a number of Lisp-related blogs, most of which are aggregated on Planet Lisp at `http://planet.lisp.org/`.

And keep your eyes peeled in all those forums for announcements of local Lisp users get-togethers in your area--in the past few years, Lispnik gatherings have popped up in cities around the world, from New York to Oakland, from Cologne to Munich, and from Geneva to Helsinki.

If you want to stick to books, here are a few suggestions. For a nice thick reference book to stick on your desk, grab *The ANSI Common Lisp Reference Book* edited by David Margolies (Apress, 2005).[13]

For more on Common Lisp's object system, you can start with *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS* by Sonya E. Keene (Addison-Wesley, 1989). Then if you really want to become an object wizard or just to stretch your mind in interesting ways, read *The Art of the Metaobject Protocol* by Gregor Kiczales, Jim des Riviéres, and Daniel G. Bobrow (MIT Press, 1991). This book, also known as AMOP, is both an explanation of what a metaobject protocol is and why you want one and the de facto standard for the metaobject protocol supported by many Common Lisp implementations.

Two books that cover general Common Lisp technique are *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp* by Peter Norvig (Morgan Kaufmann, 1992) and *On Lisp: Advanced Techniques for Common Lisp* by Paul Graham (Prentice Hall, 1994). The former provides a solid introduction to artificial intelligence techniques while teaching quite a bit about how to write good Common Lisp code, and the latter is especially good in its treatment of macros.

If you're the kind of person who likes to know how things work down to the bits, *Lisp in Small Pieces* by Christian Queinnec (Cambridge University Press, 1996) provides a nice blend of programming language theory and practical Lisp implementation techniques. While it's primarily focused on Scheme rather than Common Lisp, the same principles apply.

For folks who want a little more theoretical look at things--or who just want to know what it's like to be a freshman comp sci student at M.I.T.--*Structure and Interpretation of Computer Programs*, Second Edition, by Harold Abelson, Gerald Jay Sussman, and Julie Sussman (M.I.T. *Press, 1996) is a classic computer science text that uses Scheme to teach important programming c*oncepts. Any programmer can learn a lot from this book--just remember that there are important differences between Scheme and Common Lisp.

Once you've wrapped your mind around Lisp, you may want to place it in a bit of context. Since no one can claim to really understand object orientation who doesn't know something about Smalltalk, you might want to start with *Smalltalk-80: The Language* by Adele Goldberg and David Robson (Addison Wesley, 1989), the standard introduction to the core of Smalltalk. After that, *Smalltalk Best Practice Patterns* by Kent Beck (Prentice Hall, 1997) is full of good advice aimed at Smalltalkers, much of which is applicable to any object-oriented language.

And at the other end of the spectrum, *Object-Oriented Software Construction* by Bertrand Meyer (Prentice Hall, 1997) is an excellent exposition of the static language mind-set from the inventor of Eiffel,

an oft-overlooked descendant of Simula and Algol. It contains much food for thought, even for programmers working with dynamic languages such as Common Lisp. In particular, Meyer's ideas about Design By Contract can shed a lot of light on how one ought to use Common Lisp's condition system.

Though not about computers per se, *The Wisdom of Crowds: Why the Many Are Smarter Than the Few and How Collective Wisdom Shapes Business, Economies, Societies, and Nations* by James Surowiecki (Doubleday, 2004) contains an excellent answer to the question, "If Lisp's so great how come everybody isn't using it?" See the section on "Plank-Road Fever" starting on page 53.

And finally, for some fun, and to learn about the influence Lisp and Lispers have had on hacker culture, dip into (or read from cover to cover) *The New Hacker's Dictionary*, Third Edition, compiled by Eric S. Raymond (MIT Press, 1996) and based on the original *The Hacker's Dictionary* edited by Guy Steele (Harper & Row, 1983).

But don't let all these suggestions interfere with your programming--the only way to really learn a language is to use it. If you've made it this far, you're certainly ready to do that. Happy hacking!

---

[1]The combination of Common Lisp's read-time conditionalization and macros makes it quite feasible to develop portability libraries that do nothing but provide a common API layered over whatever API different implementations provide for facilities not specified in the language standard. The portable pathname library from Chapter 15 is an example of this kind of library, albeit to smooth over differences in interpretation of the standard rather than implementation-dependent APIs.

[2]A Foreign Function Interface is basically equivalent to JNI in Java, XS in Perl, or the extension module API in Python.

[3]As of this writing, the two main drawbacks of UFFI are the lack of support for callbacks from C into Lisp, which many but not all implementations' FFIs support, and the lack of support for CLISP, whose FFI is quite good but different enough from the others as to not fit easily into the UFFI model.

[4]Knuth has used the saying several times in publications, including in his 1974 ACM Turing Award paper, "Computer Programming as an Art," and in his paper "Structured Programs with goto Statements." In his paper "The Errors of TeX," he attributes the saying to C.A.R. Hoare. And Hoare, in an 2004 e-mail to Hans Genwitz of phobia.com, said he didn't remember the origin of the saying but that he might have attributed it to Dijkstra.

[5]CL-PPCRE also takes advantage of another Common Lisp feature I haven't discussed, *compiler macros*. A compiler macro is a special kind of macro that's given a chance to optimize calls to a specific function by transforming calls to that function into more efficient code. CL-PPCRE defines compiler macros for its functions that take regular expression arguments. The compiler macros optimize calls to those functions in which the regular expression is a constant value by parsing the regular expression at compile time rather than leaving it to be done at runtime. Look up `DEFINE-COMPILER-MACRO` in your favorite Common Lisp reference for more information about compiler macros.

[6]The word *premature* in "premature optimization" can pretty much be defined as "before profiling." Remember that even if you can speed up a piece of code to the point where it takes literally no time to run, you'll still speed up your program only by whatever percentage of time it spent in that piece of code.

[7]Declarations can appear in most forms that introduce new variables, such as `LET`, `LET*`, and the `DO` family of looping macros. `LOOP` has its own syntax for declaring the types of loop variables. The special operator `LOCALLY`, mentioned in Chapter 20, does nothing but create a scope in which you can make declarations.

[8]The FASL files produced by `COMPILE-FILE` are implementation dependent and may or may not be compatible between different versions of the same Common Lisp implementation. Thus, they're not a very good way to distribute Lisp code. The one time they can be handy is as a way of providing patches to be applied to an application running in a known version of a particular implementation. Applying the patch simply entails `LOAD`ing the FASL, and

because a FASL can contain arbitrary code, it can be used to upgrade existing data as well as to provide new code definitions.

[9]ASDF was originally written by Daniel Barlow, one of the SBCL developers, and has been included as part of SBCL for a long time and also distributed as a stand-alone library. It has recently been adopted and included in other implementations such as OpenMCL and Allegro.

[10]On Windows, where there are no symbolic links, it works a little bit differently but roughly the same.

[11]Another tool, ASDF-INSTALL, builds on top of ASDF and MK:DEFSYSTEM, providing an easy way to automatically download and install libraries from the network. The best starting point for learning about ASDF-INSTALL is Edi Weitz's "A tutorial for ASDF-INSTALL" (`http:// www.weitz.de/asdf-install/`).

[12]SLIME incorporates an Elisp library that allows you to automatically jump to the HyperSpec entry for any name defined in the standard. You can also download a complete copy of the HyperSpec to keep locally for offline browsing.

[13]Another classic reference is *Common Lisp: The Language* by Guy Steele (Digital Press, 1984 and 1990). The first edition, a.k.a. CLtL1, was the de facto standard for the language for a number of years. While waiting for the official ANSI standard to be finished, Guy Steele--who was on the ANSI committee--decided to release a second edition to bridge the gap between CLtL1 and the eventual standard. The second edition, now known as CLtL2, is essentially a snapshot of the work of the standardization committee taken at a particular moment in time near to, but not quite at, the end of the standardization process. Consequently, CLtL2 differs from the standard in ways that make it not a very good day-to-day reference. It is, however, a useful historical document, particularly because it includes documentation of some features that were dropped from the standard before it was finished as well as commentary that isn't part of the standard about why certain features are the way they are.

Copyright Š 2003-2005, Peter Seibel